

VAX/VMS System Services Reference Manual

Order No. AA-Z501C-TE

digital
software

3.3.2	Using System Services to Affect a Rights Database	3-6
3.3.2.1	Translating Identifier Names and Binary Values	3-7
3.3.2.2	Adding Identifiers and Holders to Rights Database	3-8
3.3.2.3	Determining Holders of Identifiers	3-9
3.3.2.4	Determining Identifiers Held	3-9
3.3.2.5	Modifying the Identifier Record	3-12
3.3.2.6	Modifying a Holder Record	3-12
3.3.2.7	Removing Identifiers and Holders from the Rights Database	3-14
3.3.3	Searching Operations	3-14
3.4	CREATING, TRANSLATING, AND MAINTAINING ACEs	3-16
3.4.1	Format of ACEs Types	3-17
3.4.1.1	Alarm ACE	3-17
3.4.1.2	Application ACE	3-18
3.4.1.3	Default Protection ACE	3-19
3.4.1.4	Identifier ACE	3-20
3.4.2	Translating ACEs	3-22
3.4.3	Creating and Maintaining ACEs	3-23
3.5	MODIFYING A RIGHTS LIST	3-26
3.6	CHECKING ACCESS PROTECTION	3-27
3.6.1	SYSSCHKPRO	3-27
3.6.2	SYSSCHECK_ACCESS	3-28
3.7	ADDITIONAL SECURITY SERVICES	3-28.1

SECTION 4 EVENT FLAG SERVICES 4-1

4.1	EVENT FLAG NUMBERS AND EVENT FLAG CLUSTERS	4-2
4.1.1	Specifying Event Flag and Event Flag Cluster Numbers	4-2
4.2	EXAMPLES OF EVENT FLAG SERVICES	4-3
4.2.1	Event Flag Waits	4-3
4.3	SETTING AND CLEARING EVENT FLAGS	4-4
4.4	COMMON EVENT FLAG CLUSTERS	4-5
4.5	DISASSOCIATING AND DELETING COMMON EVENT FLAG CLUSTERS	4-6

Contents

4.6	EXAMPLE OF USING A COMMON EVENT FLAG CLUSTER	4-6
4.7	COMMON EVENT FLAG CLUSTERS IN SHARED MEMORY	4-8
4.7.1	Cluster Name _____	4-8
4.8	EXAMPLE OF USING EVENT FLAG SERVICES	4-9
SECTION 5 AST (ASYNCHRONOUS SYSTEM TRAP) SERVICES		5-1
5.1	ACCESS MODES FOR AST EXECUTION	5-2
5.2	ASTS AND PROCESS WAIT STATES	5-2
5.2.1	Event Flag Waits _____	5-3
5.2.2	Hibernation _____	5-3
5.2.3	Resource Waits and Page Faults _____	5-3
5.3	HOW ASTS ARE DECLARED	5-3
5.4	THE AST SERVICE ROUTINE	5-4
5.5	AST DELIVERY	5-5
5.6	EXAMPLE OF USING AST SERVICES	5-6
SECTION 6 LOGICAL NAME SERVICES		6-1
6.1	LOGICAL NAME CONCEPTS	6-1
6.1.1	Logical Names and Equivalence Names _____	6-1
6.1.2	Logical Name Tables _____	6-2
6.1.2.1	Logical Name Directory Tables • 6-3	
6.1.2.2	Default Logical Name Tables • 6-3	
6.1.2.3	User-Defined Logical Name Tables • 6-6	
6.1.3	Privileges _____	6-6
6.1.4	Attributes _____	6-7
6.1.5	Logical Name Table Quotas _____	6-8
6.1.5.1	Directory Table Quotas • 6-8	
6.1.5.2	Default Logical Name Table Quotas • 6-9	
6.1.5.3	User-Defined Logical Name Table Quotas • 6-9	

6.1.6	Logical Name and Equivalence Name Format Conventions	6-9
6.1.7	Specifying the Logical Name Table Search List	6-10
6.2	CREATING A LOGICAL NAME—\$CRELNM	6-11
6.2.1	Duplication of Logical Names	6-12
6.2.1.1	Logical Name Supersession • 6-14	
6.3	CREATING LOGICAL NAME TABLES—\$CRELNT	6-14
6.3.1	Shareable and Named Shareable Logical Name Tables	6-15
6.3.2	\$CRELNT System Service Call	6-15
6.4	DELETING LOGICAL NAMES—\$DELLNM	6-15
6.5	TRANSLATING LOGICAL NAMES—\$STRNLNM	6-16
6.6	EXAMPLE OF USING THE LOGICAL NAME SYSTEM SERVICES	6-17

SECTION 7 INPUT/OUTPUT SERVICES 7-1

7.1	QUOTAS, PRIVILEGES, AND PROTECTION	7-2
7.1.1	Buffered I/O Quota	7-3
7.1.2	Buffered I/O Byte Count Quota	7-3
7.1.3	Direct I/O Quota	7-3
7.1.4	AST Quota	7-3
7.1.5	Physical I/O Privilege	7-4
7.1.6	Logical I/O Privilege	7-4
7.1.7	Mount Privilege	7-4
7.1.8	Volume Protection	7-4
7.1.9	Device Protection	7-6
7.1.10	System Privilege	7-6
7.1.11	Bypass Privilege	7-6
7.2	SUMMARY OF VAX/VMS QIO OPERATIONS	7-6
7.3	PHYSICAL, LOGICAL, AND VIRTUAL I/O	7-6
7.3.1	Physical I/O Operations	7-7
7.3.2	Logical I/O Operations	7-7
7.3.3	Virtual I/O Operations	7-7

Contents

7.4	I/O FUNCTION ENCODING	7-11
7.4.1	Function Codes	7-11
7.4.2	Function Modifiers	7-12
7.5	ASSIGNING CHANNELS	7-13
7.6	QUEUING I/O REQUESTS	7-13
7.7	SYNCHRONIZING SERVICE COMPLETION	7-14
7.7.1	Recommended Method for Testing Asynchronous Completion	7-16
7.8	SYNCHRONOUS FORMS OF INPUT/OUTPUT SERVICES	7-17
7.9	I/O COMPLETION STATUS	7-17
7.10	DEASSIGNING I/O CHANNELS	7-18
7.11	COMPLETE TERMINAL I/O EXAMPLE	7-18
7.12	CANCELING I/O REQUESTS	7-20
7.13	DEVICE ALLOCATION	7-20
7.13.1	Implicit Allocation	7-22
7.13.2	Deallocation	7-22
7.14	MOUNTING AND DISMOUNTING VOLUMES	7-22
7.14.1	Calling the \$MOUNT System Service	7-22
7.14.2	Calling the \$DISMOU System Service	7-24
7.15	LOGICAL NAMES AND PHYSICAL DEVICE NAMES	7-24
7.15.1	Device Name Defaults	7-25
7.16	OBTAINING INFORMATION ABOUT PHYSICAL DEVICES	7-25
7.17	FORMATTING OUTPUT STRINGS	7-27
7.18	MAILBOXES	7-28
7.18.1	Mailbox Name Format	7-31

7.18.2	System Mailboxes	7-32
7.18.3	Mailboxes for Process Termination Messages	7-33
7.18.4	Mailboxes for System Processes	7-33
7.19	EXAMPLE OF USING I/O SERVICES	7-34

SECTION 8 PROCESS CONTROL SERVICES 8-1

8.1	SUBPROCESSES AND DETACHED PROCESSES	8-2
8.2	THE EXECUTION CONTEXT OF A PROCESS	8-2
8.3	PROCESS CREATION	8-2
8.3.1	Defining an Image for a Subprocess to Execute	8-3
8.3.2	Input, Output, and Error Devices for Subprocesses	8-3
8.3.3	Disk and Directory Defaults for Created Processes	8-5
8.3.4	Controlling Resources of Created Processes	8-6
8.3.5	Detached Processes	8-7
8.4	INTERPROCESS CONTROL AND COMMUNICATION	8-7
8.4.1	Restrictions on Process Creation and Control	8-7
8.4.2	Process Identification	8-7
8.4.2.1	Process Naming Within Groups	8-8
8.4.2.2	Obtaining Information About Processes	8-9
8.4.2.3	Techniques for Interprocess Communication	8-9
8.5	PROCESS HIBERNATION AND SUSPENSION	8-11
8.5.1	Process Hibernation	8-11
8.5.2	Alternate Methods of Hibernation	8-12
8.5.3	Suspension	8-13
8.6	IMAGE EXIT	8-13
8.6.1	Image Rundown Activities	8-14
8.6.2	The \$EXIT System Service	8-14
8.6.3	Exit Handlers	8-15
8.6.4	Forced Exit	8-15
8.7	PROCESS DELETION	8-16
8.7.1	The Delete Process System Service	8-17
8.7.2	Termination Mailboxes	8-19

Contents

8.8	EXAMPLE OF USING PROCESS CONTROL SERVICES	8-21
SECTION 9 TIMER AND TIME CONVERSION SERVICES		9-1
9.1	THE SYSTEM TIME FORMAT	9-2
9.2	OBTAINING THE CURRENT DATE AND TIME	9-2
9.3	OBTAINING AN ABSOLUTE TIME IN SYSTEM FORMAT	9-3
9.4	OBTAINING A DELTA TIME IN SYSTEM FORMAT	9-3
9.5	TIMER REQUESTS	9-4
9.5.1	Canceling Timer Requests	9-6
9.6	SCHEDULED WAKEUPS	9-6
9.6.1	Canceling Scheduled Wakeups	9-6
9.7	NUMERIC AND ASCII TIME	9-7
9.8	SETTING THE SYSTEM TIME	9-7
9.9	EXAMPLE USING THE TIMER SERVICE	9-10
SECTION 10 CONDITION-HANDLING SERVICES		10-1
10.1	TYPES OF EXCEPTION	10-1
10.1.1	Change Mode and Compatibility Mode Handlers	10-6
10.2	HOW TO SPECIFY CONDITION HANDLERS	10-7
10.3	THE EXCEPTION DISPATCHER	10-7
10.4	THE ARGUMENT LIST PASSED TO A CONDITION HANDLER	10-9
10.4.1	Signal Array Arguments	10-9
10.4.2	Mechanism Array Arguments	10-10

10.5	COURSES OF ACTION FOR THE CONDITION HANDLER	10-12
10.5.1	Example of Condition-Handling Routines	10-12
10.5.2	Unwinding the Call Stack	10-13
10.6	MULTIPLE EXCEPTIONS	10-16
10.7	EXAMPLE USING CONDITION-HANDLING SERVICES	10-16
SECTION 11 MEMORY MANAGEMENT SERVICES		11-1
11.1	VIRTUAL ADDRESS SPACE	11-2
11.2	INCREASING AND DECREASING VIRTUAL ADDRESS SPACE	11-2
11.2.1	Input Address Arrays and Return Address Arrays	11-4
11.3	PAGE OWNERSHIP AND PAGE PROTECTION	11-5
11.4	WORKING SET PAGING	11-6
11.5	PROCESS SWAPPING	11-7
11.6	SECTIONS	11-8
11.6.1	Creating Sections	11-8
11.6.2	Opening the Disk File	11-9
11.6.3	Defining the Section Extents	11-10
11.6.4	Defining the Section Characteristics	11-10
11.6.5	Defining Global Section Characteristics	11-11
11.6.5.1	Global Section Name • 11-12	
11.6.6	Mapping Sections	11-14
11.6.7	Mapping Global Sections	11-15
11.6.8	Global Page-File Sections	11-16
11.6.9	Section Paging	11-16
11.6.10	Reading and Writing Data Sections	11-18
11.6.11	Releasing and Deleting Sections	11-18
11.6.12	Writing Back Sections	11-19
11.6.13	Image Sections	11-19
11.6.14	Page Frame Sections	11-19
11.7	EXAMPLE USING MEMORY MANAGEMENT SYSTEM SERVICES	11-20

SECTION 12	LOCK MANAGEMENT SERVICES	12-1
12.1	CONCEPTS OF RESOURCES AND LOCKS	12-1
12.1.1	Granularity	12-2
12.1.2	Resource Names	12-2
12.1.3	Choosing a Lock Mode	12-3
12.1.4	Levels of Locking and Compatibility	12-4
12.1.5	Lock Management Queues	12-4
12.1.6	Lock Conversion Concepts	12-6
12.1.7	Deadlock Detection	12-6
12.2	QUEUEING SIMPLE LOCK REQUESTS	12-7
12.3	ADVANCED LOCKING TECHNIQUES	12-7
12.3.1	Synchronizing Locks	12-7
12.3.2	Notification of Synchronous Completion	12-8
12.3.3	Lock Status Block	12-8
12.3.4	Blocking ASTs	12-9
12.3.5	Lock Conversions	12-9
12.3.5.1	Queueing Lock Conversions • 12-10	
12.3.6	Parent Locks	12-10
12.3.7	Lock Value Blocks	12-11
12.4	DEQUEUEING LOCKS	12-12
12.5	LOCAL BUFFER CACHING WITH THE LOCK MANAGEMENT SERVICES	12-13
12.5.1	Using the Lock Value Block	12-13
12.5.2	Using Blocking ASTs	12-14
12.5.2.1	Deferring Buffer Writes • 12-14	
12.5.2.2	Buffer Caching • 12-14	
12.5.3	Choosing a Buffer Caching Technique	12-14
12.6	EXAMPLE OF USING LOCK MANAGEMENT SERVICES	12-15

SECTION 13 PROGRAMMING EXAMPLES	13-1
13.1 ORION PROGRAM EXAMPLE	13-1
13.2 CYGNUS PROGRAM EXAMPLE	13-8
13.3 LYRA PROGRAM EXAMPLE	13-15

PART II SYSTEM SERVICE DESCRIPTIONS

\$ADD_HOLDER	SYS-1
\$ADD_IDENT	SYS-3
\$ADJSTK	SYS-5
\$ADJWSL	SYS-7
\$ALLOC	SYS-9
\$ASCEFC	SYS-12
\$ASCTIM	SYS-15
\$ASCTOID	SYS-18
\$ASSIGN	SYS-20
\$BINTIM	SYS-24
\$BRKTHRU	SYS-27
\$BRKTHRUW	SYS-35
\$CANCEL	SYS-36
\$CANEXH	SYS-38
\$CANTIM	SYS-39
\$CANWAK	SYS-41
\$CHANGE_ACL	SYS-43
\$CHECK_ACCESS	SYS-46.1
\$CHKPRO	SYS-47
\$CLREF	SYS-54
\$CMEXEC	SYS-55
\$CMKRNL	SYS-57
\$CNTREG	SYS-59
\$CRELNM	SYS-61
\$CRELNT	SYS-66
\$CREMBX	SYS-72
\$CREPRC	SYS-77
\$CREATE_RDB	SYS-91
\$CRETVA	SYS-93

Contents

\$CRMPSC	SYS-96
\$DACEFC	SYS-106
\$DALLOC	SYS-107
\$DASSGN	SYS-109
\$DCLAST	SYS-111
\$DCLCMH	SYS-113
\$DCLEXH	SYS-115
\$DELLNM	SYS-117
\$DELMBX	SYS-120
\$DELPRC	SYS-122
\$DELTVA	SYS-124
\$DEQ	SYS-126
\$DGBLSC	SYS-130
\$DISMOU	SYS-133
\$DLCEFC	SYS-136
\$ENQ	SYS-138
\$ENQW	SYS-148
\$ERAPAT	SYS-149
\$EXIT	SYS-152
\$EXPREG	SYS-153
\$FAO	SYS-155
\$FILESCAN	SYS-170
\$FIND_HELD	SYS-174
\$FIND HOLDER	SYS-177
\$FINISH_RDB	SYS-180
\$FORCEX	SYS-181
\$FORMAT_ACL	SYS-183
\$GETDVI	SYS-192
\$GETDVIW	SYS-208
\$GETJPI	SYS-209
\$GETJPIW	SYS-222.2
\$GETLKI	SYS-223
\$GETLKIW	SYS-234
\$GETMSG	SYS-235
\$GETQUI	SYS-239
\$GETQUIW	SYS-271
\$GETSYI	SYS-272
\$GETSYIW	SYS-282
\$GETTIM	SYS-283
\$GETUAI	SYS-284
\$GRANTID	SYS-292
\$HIBER	SYS-296
\$IDTOASC	SYS-298

\$LCKPAG	SYS-301
\$LKWSET	SYS-303
\$MGBLSC	SYS-305
\$MOD_HOLDER	SYS-309
\$MOD_IDENT	SYS-312
\$MOUNT	SYS-315
\$MTACCESS	SYS-320.6
\$NUMTIM	SYS-321
\$PARSE_ACL	SYS-323
\$PURGWS	SYS-325
\$PUTMSG	SYS-326
\$QIO	SYS-334
\$QIOW	SYS-339
\$READEF	SYS-340
\$REM_HOLDER	SYS-342
\$REM_IDENT	SYS-344
\$RESUME	SYS-346
\$REVOKID	SYS-348
\$SCHDWK	SYS-353
\$SETAST	SYS-356
\$SETEF	SYS-357
\$SETEXV	SYS-358
\$SETIME	SYS-360
\$SETIMR	SYS-362
\$SETPRA	SYS-364
\$SETPRI	SYS-366
\$SETPRN	SYS-368
\$SETPRT	SYS-369
\$SETPRV	SYS-372
\$SETRWM	SYS-376
\$SETSFM	SYS-378
\$SETSSF	SYS-380
\$SETSTK	SYS-382
\$SETSWM	SYS-384
\$SETUAI	SYS-385
\$SNDERR	SYS-393
\$SNDJBC	SYS-394
\$SNDJBCW	SYS-428.9
\$SNDOPR	SYS-429
\$SUSPND	SYS-443
\$SYNCH	SYS-445
SY\$SRMSRUNDWN	SYS-446.1
SY\$SETDDIR	SYS-446.3

Contents

SYS\$SETDFPROT	SYS-446.5
\$TRNLNM	SYS-447
\$ULKPAG	SYS-452
\$ULWSET	SYS-454
\$UNWIND	SYS-456
\$UPDSEC	SYS-458
\$UPDSECW	SYS-462
\$WAITFR	SYS-463
\$WAKE	SYS-464
\$WFLAND	SYS-466
\$WFLOR	SYS-468

APPENDIX A PRIVILEGED SHAREABLE IMAGES

A-1

A.1	CODING THE PRIVILEGED SHAREABLE IMAGE	A-1
A.1.1	Change-Mode Vector _____	A-2
A.1.2	Entry Point to the Privileged Shareable Image _____	A-3
A.1.3	Kernel-Mode or Executive-Mode Dispatcher _____	A-3
A.1.4	Enabling and Disabling User Privileges _____	A-3

A.2	LINKING THE PRIVILEGED SHAREABLE IMAGE	A-4
A.2.1	Specifying Protection for the Image or Clusters _____	A-4

A.3	INSTALLING THE PRIVILEGED SHAREABLE IMAGE	A-5
------------	--	------------

A.4	USING THE PRIVILEGED SHAREABLE IMAGE	A-5
------------	---	------------

A.5	PROGRAM LISTINGS	A-5
------------	-------------------------	------------

APPENDIX B USING SHARED MEMORY

B-1

B.1	PREPARING MULTIPOINT MEMORY FOR USE	B-1
------------	--	------------

B.2	PRIVILEGES REQUIRED FOR SHARED MEMORY USE	B-2
------------	--	------------

B.3	NAMING FACILITIES IN SHARED MEMORY	B-2
------------	---	------------

B.4	ASSIGNING LOGICAL NAMES AND LOGICAL NAME TRANSLATION	B-3
B.5	HOW VAX/VMS FINDS FACILITIES IN SHARED MEMORY	B-4
B.6	USING COMMON EVENT FLAGS IN SHARED MEMORY	B-5
B.7	USING MAILBOXES IN SHARED MEMORY	B-5
B.8	USING GLOBAL SECTIONS IN SHARED MEMORY	B-6
B.8.1	Removing Shared Memory Global Sections	B-7
B.8.2	Create and Map Section System Service	B-8

INDEX

FIGURES

2-1	Procedure Argument Passing Mechanisms	2-18
2-2	Interpreting MACRO Examples	2-20
7-1	Files-11 Volume Protection Fields	7-5
7-2	Foreign Volume Protection Fields	7-5
7-3	Mailbox Protection Fields	7-5
7-4	Physical I/O Access Checks	7-8
7-5	Logical I/O Access Checks	7-9
7-6	Physical, Logical, and Virtual I/O	7-10
7-7	I/O Function Format	7-11
7-8	Function Modifier Format	7-12
7-9	I/O Status Block	7-18
7-10	\$MOUNT Item Descriptor	7-23
8-1	Image Exit and Process Deletion	8-18
10-1	Search of Stack for Condition Handler	10-8
10-2	Argument List and Arrays Passed to Condition Handler	10-11
10-3	Unwinding the Call Stack	10-15
11-1	Layout of Process Virtual Address Space	11-3
12-1	Model Database	12-2
12-2	Three Lock Queues	12-5
12-3	A Deadlock	12-6
12-4	The Lock Status Block	12-8

TABLES

1-1	Main Headings in the Routine Template	1-3
3-1	ACE Type-Independent Information	3-18
4-1	Summary of Event Flag and Cluster Numbers	4-2
6-1	Summary of Privileges	6-7
7-1	Read and Write I/O Functions	7-12
7-2	Default Device Names for I/O Services	7-26
8-1	Process Identification	8-9
8-2	Process Hibernation and Suspension	8-11
10-1	Summary of Exception Conditions	10-2
11-1	Sample Virtual Address Arrays	11-5
11-2	Flag Bits to Set for Specific Section Characteristics	11-11
12-1	Compatibility of Lock Modes	12-4
12-2	Effect of Lock Conversion on Lock Value Block	12-11
SYS-5	User Privileges	SYS-78
SYS-1	Required and Optional Arguments for the \$CRMPSC Service	SYS-100
SYS-2	List of FAO Directives	SYS-158
SYS-3	FAO Output Field Lengths and Fill Characters	SYS-160
SYS-4	Process Identification in \$GETJPI	SYS-221
SYS-5	User Privileges	SYS-372
SYS-6	CPU Time Limit Decision Table	SYS-412
SYS-7	Working Set Decision Table	SYS-427

Preface

This manual provides users of the VAX/VMS operating system with detailed usage and reference information on the system services.

VAX/VMS system services can be used only in programs written in languages that produce native code for the VAX hardware. At present these languages include VAX MACRO and the following high-level languages:

- VAX BASIC
- VAX BLISS-32
- VAX C
- VAX COBOL
- VAX COBOL-74
- VAX CORAL
- VAX DIBOL
- VAX FORTRAN
- VAX PASCAL
- VAX PL/1

Intended Audience

This manual is intended for system and application programmers who want to call system services.

Structure of This Document

This manual is organized into two parts and two appendixes, as follows:

- Part I provides guidelines on the use of system services.

Section 1 introduces the system services. It presents overviews of the categories of system services and explains the documentation format of the service descriptions in Part II.

Section 2 describes how to call system services. It contains detailed information for the VAX MACRO programmer and general information for the high-level language programmer. For additional information about a high-level language and programming examples in that language, see the language's user's guide.

Sections 3 through 12 guide new users in understanding how the system services work and how to use them. Each category of services has its own section. Examples are provided in VAX MACRO and VAX FORTRAN, although they are explained in a way meaningful to all high-level language programmers.

Section 13 contains sample programs that use various system services.

- Part II provides detailed reference information on each system service. This information is presented using the documentation format described in Section 1. Service descriptions appear in alphabetical order by service name.
- Appendix A contains information on how you can code your own system services. The original version of this material appeared in the Version 3.0 VAX/VMS *Real-Time User's Guide*, Chapter 6.

- Appendix B provides a guide for programmers in their use of shared memory. The original version of this material appeared in the Version 3.0 *VAX/VMS Real-Time User's Guide*, Chapter 5.

Associated Documents

The VAX Procedure Calling and Condition Handling Standard, which is documented in Section 2.5 of the *Introduction to VAX/VMS System Routines*, contains useful information for anyone who wants to call system services.

VAX MACRO programmers will find additional information on calling system services in the *VAX MACRO and Instruction Set Reference Volume*.

High-level language programmers will find additional information on calling system services in the language reference manual and language user's guide provided with your VAX language.

The following documents may also be useful.

- *Guide to Using DCL and Command Procedures on VAX/VMS*
- *Guide to VAX/VMS File Applications*
- *Guide to VAX/VMS System Security*
- *VAX/VMS Networking Manual*
- *VAX Record Management Services Reference Manual*
- *VAX/VMS I/O User's Reference Manual: Part I*
- *VAX/VMS I/O User's Reference Manual: Part II*

For a complete list and description of the manuals in the VAX/VMS document set, see the *Introduction to the VAX/VMS Document Set*.

Conventions Used in This Document

The conventions used in this document are described in Section 1.1 of this manual, "Documentation Format for System Service Routines."

New and Changed Features

The sections that follow describe the changes that have been made to the *VAX/VMS System Services Reference Manual* for VAX/VMS Version 4.4.

Changes in Documentation Format

The tables that describe VMS usages and VAX data types are now contained in the *Introduction to VAX/VMS System Routines*. In addition, the *Introduction to VAX/VMS System Routines* contains language implementation charts, which describe how to construct each of the VMS usages in several high-level languages.

With Version 4.4, the *VAX/VMS System Services Reference Manual* no longer contains the appendix documenting system services that became obsolete at Version 4.0. If you still have programs that use any of these obsolete services, you should retain the documentation contained in Appendix A of the Version 4.0 and Version 4.2 *VAX/VMS System Services Reference Manual*.

New Services

The following new services have been added since the Version 4.2 edition of the *VAX/VMS System Services Reference Manual*.

Service	Description
\$CHECK_ACCESS	Check Access
\$GETUAI	Get User Authorization Information
\$SETUAI	Set User Authorization Information

The following services have been moved from the Version 4.2 edition of the *Record Management Services Manual*. The descriptions of these three services are now included in Part II of this manual. Because there are no predefined macros with which you can call these three services, they are listed with the full routine name, including the SYS prefix.

Service	Description
SYS\$RMSRUNDOWN	RMS Rundown
SYS\$SETDDIR	Set Default Directory
SYS\$SETDEFPROT	Set Default Protection

Modified Services

The following list describes the system services that have been modified for Version 4.4.

- For VAX/VMS Version 4.4, the DYNAMIC attribute is being added to attributes currently found in many of the security-related system services.

New and Changed Features

The following system services are affected:

\$ADD_HOLDER
\$ADD_IDENT
\$ASCTOID
\$FIND_HELD
\$FIND_HOLDER
\$GRANTID
\$IDTOASC
\$MOD_HOLDER
\$MOD_IDENT
\$REVOKID

- The \$SNDJBC service contains the following new item codes:

SJC\$_DEFAULT_FORM_NAME/SJC\$_DEFAULT_FORM_NUMBER
SJC\$_QUEMAN_RESTART/SJC\$_NO_QUEMAN_RESTART

Changes have been made in the documentation of the following item codes:

SJC\$_AFTER_TIME/SJC\$_NO_AFTER_TIME
SJC\$_FORM_NAME/SJC\$_FORM_NUMBER
SJC\$_HOLD/SJC\$_NO_HOLD
SJC\$_PRIORITY

- The \$GETQUI service contains three new item codes:

QUI\$_DEFAULT_FORM_NAME
QUI\$_DEFAULT_FORM_NUMBER
QUI\$_JOB_PID

- The \$GETJPI service contains one previously undocumented item code:

JPI\$_JOBTYPE

- The \$GETDVI service contains three previously undocumented item codes:

DVI\$_MEDIA_ID
DVI\$_MEDIA_NAME
DVI\$_MEDIA_TYPE

- The \$MOUNT service includes one new flag option:

MNT\$_TAPE_DATE_WRITE

PART I Introduction to System Services

1

Introduction to System Services

System services are procedures that the VAX/VMS operating system uses to control resources available to processes; to provide for communication among processes; and to perform basic operating system functions, such as the coordination of input/output operations.

Although most system services are used primarily by the operating system itself on behalf of logged-in users, many are available for general use and provide mechanisms that can be used in application programs. For example, when you log in to the system, the Create Process (\$CREPROC) system service is called to create a process on your behalf. You may, in turn, write a program that calls the \$CREPROC system service to create a subprocess to perform certain functions for an application.

System services can be divided into functional groups. The following table lists each group of system services and its function.

Services Group	Function
Security	The security services provide various mechanisms that you can use to enhance the security of VAX/VMS systems.
Event Flag	A process can use event flags to synchronize sequences of operations in a program. Event flag services clear, set, and read event flags, and place a process in a wait state pending the

Contents

PREFACE	xix
NEW AND CHANGED FEATURES	xxi

PART I INTRODUCTION TO SYSTEM SERVICES

SECTION 1 INTRODUCTION TO SYSTEM SERVICES	1-1
---	-----

1.1 DOCUMENTATION FORMAT FOR SYSTEM SERVICE ROUTINES	1-3
1.1.1 Format Heading	1-4

Introduction to System Services

Services Group	Function
Input/Output	<p>I/O services perform input and output operations directly, rather than through the file handling services of the VAX Record Management Services (RMS). I/O services do the following:</p> <ul style="list-style-type: none"> • Perform logical, physical, and virtual input/output operations • Format output lines converting binary numeric values to ASCII strings and substituting variable data in ASCII strings • Perform network operations • Queue messages to system processes
Process Control	Process control services allow you to create, delete, and control the execution of processes.
Timer and Time Conversion	Timer services schedule program events for a particular time of the day, or after a specified interval of time has elapsed. The time conversion services provide a way to obtain and format binary time values for use with the timer services.
Condition-Handling	Condition handlers are procedures that can be designated to receive control when a hardware or software exception condition occurs during image execution. Condition-handling services designate condition handlers for special purposes.
Memory Management	<p>Memory management services provide ways to use the virtual address space available to a program. Included are services that do the following:</p> <ul style="list-style-type: none"> • Allow an image to increase or decrease the amount of virtual memory available • Control the paging and swapping of virtual memory • Create and access files in memory that contain shareable code or data
Change Mode	Change mode services alter the access mode of a process to a more privileged mode to execute particular routines, or change the stack pointer for a less privileged mode. These services are used primarily by the operating system.
Lock Management	Lock management services allow cooperating processes to synchronize their access to shared resources.

2.6	CONDITION VALUES RETURNED FROM SYSTEM SERVICES	2-14
2.6.1	Information Provided by Condition Values	2-15
2.6.2	Testing Return Condition Values	2-15

Introduction to System Services

Documentation Format for System Service Routines

1.1

Documentation Format for System Service Routines

Each system service routine in Part II of this book is documented using a structured format called the routine template. This section discusses the main headings in the routine template, the information that is presented under each heading, and the format used to present the information.

The purpose of this section, therefore, is to explain where to find information and how to read it correctly, not how to use it. See the *Introduction to VAX/VMS System Routines* for substantive discussion of the contents, meaning, and use of the information provided in the routine template.

Some main headings in the routine template contain information that requires no further explanation beyond what is given in Table 1-1. However, the following main headings contain information that does require additional discussion, and this discussion takes place in the remaining subsections of this section.

Format Heading
Returns Heading
Arguments Heading
Condition Values Returned Heading

Table 1-1 Main Headings in the Routine Template

Main Heading	Description
Routine Name	Required. The routine entry point name appears at the top of the first page. It is usually, though not always, followed by the English text name of the routine.
Routine Overview	Required. The routine overview appears directly below the routine name; the overview explains, usually in one or two sentences, what the routine does.
Format	Required. The format heading follows the routine overview. The format gives the routine entry point name and the routine argument list.
Returns	Required. The returns heading follows the routine format. It explains what information is returned by the routine.
Arguments	Required. The arguments heading follows the returns heading. Detailed information about each argument is provided under the arguments heading. If a routine takes no arguments, the word "None" appears.
Description	Optional. The description heading follows the arguments heading. The description section contains information about specific actions taken by the routine: interaction between routine arguments, if any; operation of the routine within the context of VAX/VMS; user privileges needed to call the routine, if any; system resources used by the routine; and user quotas that may affect the operation of the routine. For some simple routines, a description section is not necessary because the routine overview carries the needed information.

Introduction to System Services

Documentation Format for System Service Routines

Table 1-1 (Cont.) Main Headings in the Routine Template

Main Heading	Description
Condition Values Returned	Required. The condition values returned section appears following the description section. It lists the condition values (typically status or completion codes) that are returned by the routine.

1.1.1 Format Heading

Under the format heading, the following two types of information can be present.

- Procedure call format
- Explanatory text

All system service routines have a procedure call format. Use of the procedure call format results in a routine call conforming to the procedure call mechanism described in the VAX Procedure Calling and Condition Handling Standard; for example, an entry mask is created, registers are saved, and so on.

Explanatory text may appear following the procedure call format. This text is present only when needed to clarify the format(s). For example, the call format indicates that arguments are optional by enclosing them in brackets ([]). However, brackets alone cannot convey all the important information that may apply to optional arguments. For example, in some routines that have many optional arguments, if one optional argument is selected, another optional argument must also be selected. In such cases, text following the format clarifies this fact.

A procedure call format appears under the format heading as follows. The format given here, though intended to be generic, is in fact specific to some extent; it is chosen in order to bring to light some of the syntactical mechanisms used to handle the more complex routine calls.

ENTRY-POINT-NAME **arg1 ,arg2 [,arg3] ,nullarg [,arg4] [,arg5]**

The sample format exemplifies the use of the following syntax rules.

Element	Syntax Rule
Entry point names	Entry point names are always shown in uppercase characters.
Argument names	Argument names are always shown in lowercase characters.
Spaces	One or more spaces are used between the entry point name and the first argument, and between arguments.
Brackets ([])	Brackets surround optional arguments; arg3 , arg4 , and arg5 are optional arguments because they are surrounded by brackets. Note that commas too can be optional (see the comma element).

Introduction to System Services

Documentation Format for System Service Routines

Element	Syntax Rule
Commas	<p>Between arguments, the comma always follows the space. If the argument is optional, the comma may appear inside the brackets or outside the brackets, depending on the position of the argument in the list and on whether surrounding arguments are optional or required.</p> <p>For example, arg3 in the sample format is an optional argument; but because there are other required arguments that follow arg3 in the list, the comma itself is not optional (since it marks the place of arg3); therefore, the comma is not inside the brackets.</p> <p>The arguments arg4 and arg5 are optional. Because there are no required arguments that follow arg4 and arg5 in the list, the commas in front of arg4 and arg5 are themselves optional; that is, they (the commas) would not be specified in the call if arg4 and arg5 were not specified. Therefore, the commas in front of arg4 and arg5 are inside the brackets.</p>
Null arguments	<p>A null argument is a place-holding argument. It is used to hold a place in the argument list for an argument that has not yet been implemented by DIGITAL but which may be at some future time. A null argument is always given the name "null_arg."</p> <p>When calling a routine that has a null argument, you must either: (1) supply the value 0 for the null argument or (2) supply no value but include the comma in the call format to mark its place.</p>

1.1.2 Returns Heading

Under the returns heading appears information that describes what information, if any, is returned by the routine to the caller. For system services, the information that is returned is always a longword condition value.

This condition value contains various kinds of information, but most importantly for the caller, it describes (in bits 0 through 3) the completion status of the operation. Programmers test the condition value to determine if the routine completed successfully.

For the purposes of high-level language programmers, the fact that status information is returned by means of a condition value and that it is returned in a VAX register (R0) is of little importance because the high-level language programmer receives this status information in the return (or status) variable he or she uses when making the call. The run-time environment established for the high-level language program allows the status information in R0 to be moved automatically to the user's return variable.

Note that the *Condition Values Returned* heading in the routine template describes the possible condition values that the routine can return.

Introduction to System Services

Documentation Format for System Service Routines

1.1.3 Arguments Heading

Under the arguments heading appears detailed information about each argument listed in the call format. Arguments are described in the order in which they appear in the call format. If the routine has no arguments, the term "none" appears.

The following format is used to describe each argument.

argument-name
VMS Usage: **argument-VMS-data-type**
type: **argument-data-type**
access: **argument-access**
mechanism: **argument-passing-mechanism**

One paragraph of structured text is followed by other paragraphs of text, as needed.

1.1.3.1 VMS Usage Entry

The VMS usage entry indicates the VMS data type of the argument. Each VMS data type has one and only one storage representation: for example, the VMS data type "access_mode" is an unsigned byte. In addition, a VMS data type may or may not have a "conceptual" meaning.

Most VMS data types may be considered as "conceptual" types; that is, they carry meaning that is unique in the context of the VMS operating system. Take, for example, the VMS type "access_mode". The storage representation of this VMS type is an unsigned byte, and the conceptual content of this unsigned byte rests in the fact that it designates a hardware access mode and has therefore only four valid values: 0, designating kernel mode; 1, executive mode; 2, supervisor mode; and 3, user mode. However, some VMS data types are not conceptual types; that is, they specify a storage representation, but carry no other semantic content from the point of view of VAX/VMS: for example, the VMS type "byte_signed" is not a conceptual type.

The VMS data types are described in full in the *Introduction to VAX/VMS System Routines*. The *Introduction to VAX/VMS System Routines* also contains language implementation charts, which describe how to construct each of the VMS data types in a number of high-level languages.

1.1.3.2 Type Entry

When a calling program passes an argument to a system service, the service expects the argument to be of a particular data type. The service descriptions in Part II indicate the expected data types for each argument.

Properly speaking, an argument does not have a data type; rather, the data specified by an argument has a data type. The argument is merely the vehicle for the passing of data to the called routine.

As described in the VAX Procedure Calling Standard in *Introduction to VAX/VMS System Routines*, procedure calls result in the construction of an *argument list*. This argument list is a vector of longwords. The first longword on the list contains a count of the number of remaining longwords, and each remaining longword is one argument. Thus, an *argument* is one longword in the argument list.

Nevertheless, the phrase "argument data type" is frequently used to describe the data type of the data that is specified by the argument. This terminology is used because it is simpler and more straightforward than the strictly accurate phrase "data type of the data specified by the argument".

Introduction to System Services

Documentation Format for System Service Routines

The *Introduction to VAX/VMS System Routines* describes the data types allowed by the VAX Procedure Calling Standard.

1.1.3.3

Access Entry

The argument-access entry describes the way in which the called routine accesses the data specified by the argument. The following three methods of access are the most common.

- 1 Read only. Data upon which a routine operates, or data needed by the routine to perform its operation, must be *read* by the called routine. Such data is also called *input* data. When an argument specifies input data, the "access" entry shows "read only".

The term "only" is present to indicate that the called routine does not both read and write (that is, "modify") the input data. Thus, input data supplied by a variable is preserved when the called routine completes execution.

- 2 Write only. Data that the called routine returns to the calling routine must be *written* into a location where the calling routine can access it. Such data is also called *output* data. When an argument specifies output data, the "access" entry shows "write only".

The term "only" is present to indicate that the called routine does not read the contents of the location either before or after it writes into the location.

- 3 Modify. When an argument specifies data that is both read and written by the called routine, the "access" entry shows "modify". In this case, the called routine reads the input data, which it uses in its operation, and then overwrites the input data with the results (the output data) of the operation. Thus, when the called routine completes execution, the input data specified by the argument is lost.

The following is a complete list of the access types allowed by the VAX Procedure Calling Standard.

- Read only
- Write only
- Modify
- Function call (before return)
- JMP after unwind
- Call after stack unwind
- Call without stack unwind

1.1.3.4

Mechanism Entry

The way in which an argument specifies the actual data to be used by the called routine is defined in terms of the argument passing mechanism. There are three types of passing mechanism.

- 1 By value. When the longword argument in the argument list contains the actual data to be used by the routine, the actual data is said to be passed to the routine "by value". In this case, the longword argument contains the actual data; in other words, the argument is the actual data. Note that since an argument is only one longword in length, only data that can be represented in one longword can be passed by value.

Introduction to System Services

Documentation Format for System Service Routines

- 2 By reference. When the longword argument in the argument list contains the address of the data to be used by the routine, the data is said to be passed "by reference". In this case, the argument is a pointer to the data.
- 3 By descriptor. When the longword argument in the argument list contains the address of a descriptor, the data is said to be passed "by descriptor". A descriptor consists of two or more longwords (depending on the type of descriptor used), which describe the location, length, and data type of the data to be used by the called routine. In this case, the argument is a pointer to a descriptor that itself is a pointer to the actual data.

The following list contains the passing mechanisms allowed by the VAX Procedure Calling Standard.

Passing Mechanism	Descriptor Code
By value	
By reference	
By reference, array reference	
By descriptor	
By descriptor, fixed-length	DSC\$K_CLASS_S
By descriptor, dynamic string	DSC\$K_CLASS_D
By descriptor, array	DSC\$K_CLASS_A
By descriptor, procedure	DSC\$K_CLASS_P
By descriptor, decimal string	DSC\$K_CLASS_SD
By descriptor, noncontiguous array	DSC\$K_CLASS_NCA
By descriptor, varying string	DSC\$K_CLASS_VS
By descriptor, varying string array	DSC\$K_CLASS_VSA
By descriptor, unaligned bit string	DSC\$K_CLASS_UBS
By descriptor, unaligned bit array	DSC\$K_CLASS_UBA
By descriptor, string with bounds	DSC\$K_CLASS_SB
By descriptor, unaligned bit string with bounds	DSC\$K_CLASS_UBSB

1.1.3.5 Explanatory Text Entry

For each argument, one or more paragraphs of explanatory text follows the "type", "access", and "mechanism" entries. The first paragraph is highly structured and always contains the following items of information.

- 1 An initial sentence fragment that describes: (1) the nature of the data specified by the argument and (2) the way in which the routine uses this data. For example, if an argument were supplying a number, which the routine was to convert to another data type, the initial sentence fragment would be something like the following: "number that is to be converted to the such-and-such data type."
- 2 A sentence expressing the relationship between the argument and the data that it specifies. This relationship is the passing mechanism used to pass the data.

If the passing mechanism is "by value", this sentence says something like the following: "the xxx argument contains (or is) the such-and-such data."

Introduction to System Services

Documentation Format for System Service Routines

If the passing mechanism is "by reference", this sentence says something like the following: "the xxx argument is the address of the such-and-such data."

If the passing mechanism is "by descriptor", this sentence says something like the following: "the xxx argument is the address of a descriptor pointing to the such-and-such data."

Additional explanatory paragraphs appear for each argument as needed. For example, some arguments specify complex data consisting of many discrete fields, each of which has a particular purpose and use. In such cases, additional paragraphs provide detailed descriptions of each such field, symbolic names for the fields, if any, and guidance relating to their use.

1.1.4 Condition Values Returned Heading

A condition value is an unsigned longword that has several uses in the VAX architecture.

- It indicates the success or failure of a called procedure.
- It describes an exception condition when an exception is signaled.
- It identifies system messages.
- It reports program success or failure to the command language level.

Figure 2-2 in the *Introduction to VAX/VMS System Routines* depicts the format and contents of the longword condition value, and Section 1.2.5 in the *Introduction to VAX/VMS System Routines* describes these contents and explains in detail the uses of the condition value.

Under the *Condition Values Returned* heading, a two-column list gives the symbolic code for each condition value that the routine can return and its accompanying description. This description explains whether the condition value indicates success or failure, and if failure, what user action may have caused the failure and what can be done to correct it.

Symbolic codes for condition values are system defined. The symbolic code defined for each condition value equates to a number that is identical to the longword condition value when interpreted as a number. In other words, though the condition value consists of several fields, each of which can be interpreted individually for specific information, the entire longword condition value itself can be interpreted as an unsigned longword integer, and this integer has an equivalent symbolic code.

Note that if a called routine generates an exception condition during execution, the exception condition is *signaled*; the exception condition is then *handled* by a condition handler (either user-supplied or system-supplied). Depending on the nature of the exception condition and on the condition handler that handles the exception condition, the called routine will either continue normal execution or terminate abnormally.

The documentation heading *Condition Values Returned* describes the condition values returned by the routine when it completes execution without generating an exception condition.

Introduction to System Services

Documentation Format for System Service Routines

1.1.5 Condition Values Returned in the I/O Status Block Heading

When the called routine returns a condition value in an I/O status block, the possible condition values that the routine can return are listed under the "Condition Values Returned in the I/O Status Block" heading.

The routines that return condition values in the I/O status block are the system services that complete asynchronously.

Some system services complete asynchronously; that is, they return to the caller immediately after the call to the service is successfully queued but before the operation to be performed by the service has completed. This allows the calling program to continue execution while the system service itself is executing. System services that complete asynchronously all have arguments that specify an I/O status block. When the system service operation has completed, a condition value specifying the completion status of the operation is written to the I/O status block.

The first word in the I/O status block receives the condition value for the final completion status of an asynchronous system service. Representing a longword condition value in a word-length field is possible for system services because the high-order word in system service condition values is zero.

Section 1.2.5 in the *Introduction to VAX/VMS System Routines* explains the contents of the fields in the longword condition value in detail. However, the reason why a system service condition value can be represented in one word, rather than one longword, is worth mentioning here.

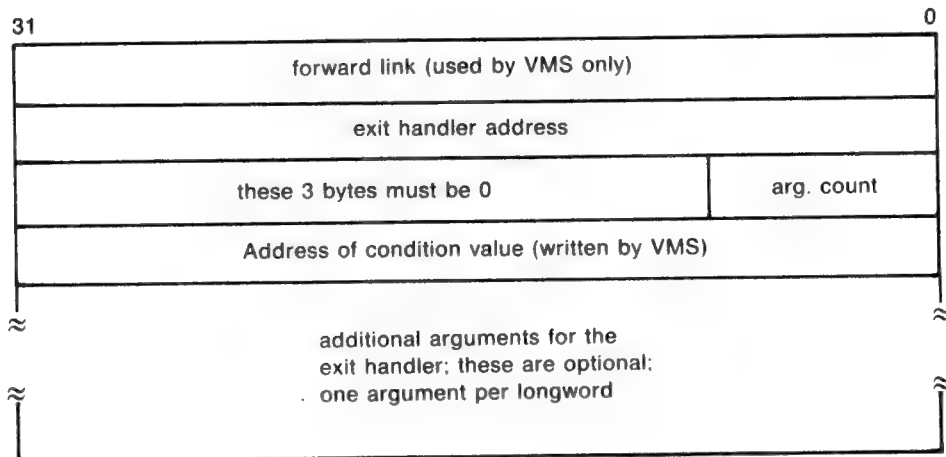
One field in the condition value specifies which facility generated the condition value; this field is in the high-order word of the longword condition value. For the system facility, the value of this field is zero. This fact allows condition values generated by the system facility (which includes all system services) to be represented in a word, rather than a longword, since bits in the high-order word are all zeros.

Introduction to System Services

Documentation Format for System Service Routines

Table 1-2 (Cont.) VMS Data Structures

Data Structure	Definition
date_time	64-bit unsigned, binary integer denoting a date and time as the number of elapsed 100-nanosecond units since 00:00 o'clock, November 17, 1858. This VMS data type is the same as the data type "absolute date and time" in Table 1-3.
device_name	Character string denoting the 1 to 15-character name of a device. It can be a logical name, but if it is, it must translate to a valid device name. If the device name contains a colon (:), the colon and the characters past it are ignored. When an underscore (_) precedes device name string, it indicates that the string is a physical device name.
ef_cluster_name	Character string denoting the 1 to 15-character name of an event flag cluster. It can be a logical name, but if it is, it must translate to a valid event flag cluster name. For more information on how the system translates logical names to event flag cluster names see the "Event Flag Services" section of the <i>VAX/VMS System Services Reference Manual</i> .
ef_number	Unsigned longword integer denoting the number of an event flag. Local event flags numbered 32 to 63 are available to your programs.
exit_handler_block	Variable-length structure denoting an exit handler control block. This control block, which describes the exit handler, is depicted in the following diagram.



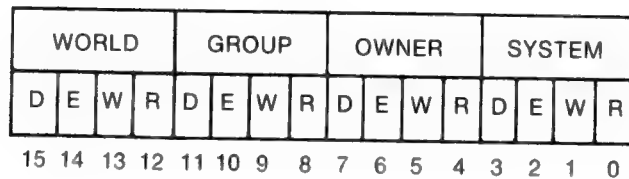
ZK-1714-84

Introduction to System Services

Documentation Format for System Service Routines

Table 1-2 (Cont.) VMS Data Structures

Data Structure	Definition
fab	Structure denoting an RMS file access block. A complete description of this structure is contained in the <i>VAX Record Management Services Reference Manual</i> .
file_protection	<p>Unsigned word that is a 16-bit mask that specifies file protection. The mask contains four 4-bit fields, each of which specifies the protection to be applied to file access attempts by one of the four categories of user: from the rightmost field to the leftmost field, (1) system users, (2) the file owner, (3) users in the same UIC group as the owner, and (4) all other users (the world). Each field specifies, from the rightmost bit to the leftmost bit: (1) read access, (2) write access, (3) execute access, (4) delete access. Set bits indicate that access is denied.</p> <p>The following diagram depicts the 16-bit file-protection mask.</p>

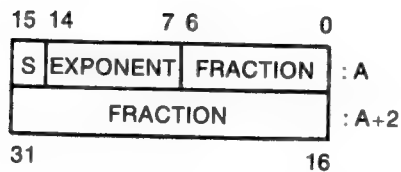


ZK-1706-84

floating_point

One of the VAX standard floating-point data types. These types are F_floating, D_floating, G_floating, and H_floating.

The structure of an F_floating number is as follows:



ZK-4197-85

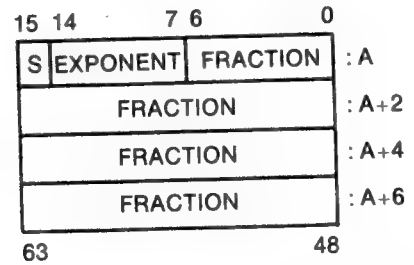
Introduction to System Services

Documentation Format for System Service Routines

Table 1-2 (Cont.) VMS Data Structures

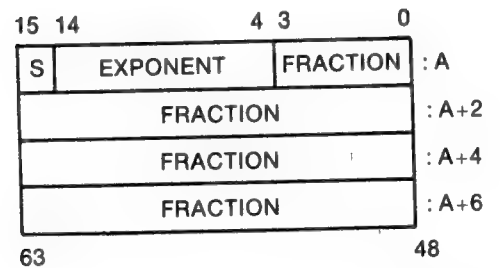
Data Structure	Definition
----------------	------------

The structure of a D_floating number is as follows:



ZK-4198-85

The structure of a G_floating number is as follows:



ZK-4199-85

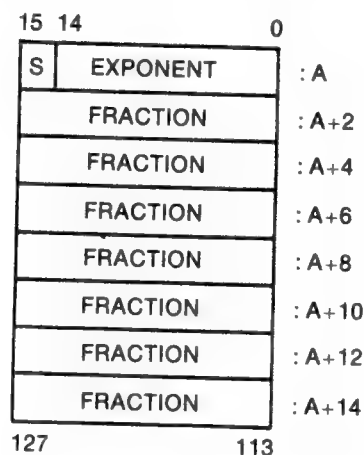
Introduction to System Services

Documentation Format for System Service Routines

Table 1-2 (Cont.) VMS Data Structures

Data Structure	Definition
----------------	------------

The structure of an H-floating number is as follows:



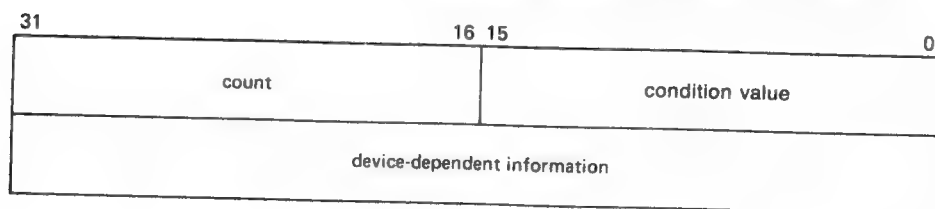
ZK-4196-85

function_code

Unsigned longword specifying the exact operations a procedure is to perform. This longword has two word-length fields: the first field is a number specifying the major operation; the second field is a mask or bit vector specifying various suboperations within the major operation.

io_status_block

Quadword structure containing information returned by a procedure that completes asynchronously. The information returned varies depending on the procedure. The following figure illustrates the format of the information written in the IOSB for SYS\$QIO.



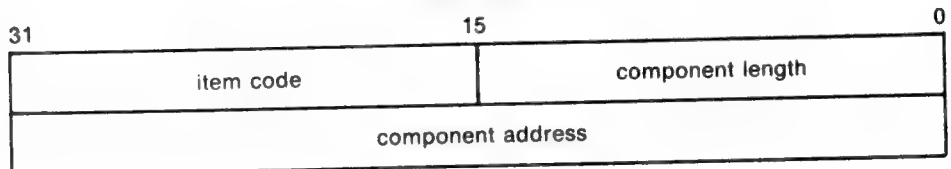
ZK-856-82

Introduction to System Services

Documentation Format for System Service Routines

Table 1-2 (Cont.) VMS Data Structures

Data Structure	Definition
	<p>The first word contains a condition value indicating the success or failure of the operation. The condition values used are the same as for all returns from system services; for example, <code>SS\$_NORMAL</code> indicates successful completion.</p> <p>The second word contains the number of bytes actually transferred in the I/O operation. Note that for some devices this word contains only the low-order word of the count. For information on specific devices, see the <i>VAX/VMS I/O Reference Volume</i>.</p> <p>The second longword contains device-dependent return information.</p> <p>To ensure successful I/O completion and the integrity of data transfers, the IOSB should be checked following I/O requests, particularly for device-dependent I/O functions. For complete details on how to use the I/O status block, see the <i>VAX/VMS I/O Reference Volume</i>.</p>
item_list_2	<p>Structure that consists of one or more item descriptors and that is terminated by a longword containing 0. Each item descriptor is a 2-longword structure that contains three fields. The following diagram depicts a single item descriptor:</p>



ZK-1709-84

The first field is a word in which the service writes the length (in characters) of the requested component. If the service does not locate the component, it returns the value 0 in this field and in the component address field.

The second field contains a user-supplied, word-length symbolic code that specifies the component desired. The item codes are defined by the macros that are specific to the service.

The third field is a longword in which the service writes the starting address of the component. This address is within the input string itself.

Introduction to System Services

Documentation Format for System Service Routines

Table 1-2 (Cont.) VMS Data Structures

Data Structure	Definition						
item_list_3	Structure that consists of one or more item descriptors and that is terminated by a longword containing 0. Each item descriptor is a 3-longword structure that contains four fields. The following diagram depicts the format of a single item descriptor.						
<div style="display: flex; justify-content: space-between; width: 100%;"> 31 15 0 </div> <table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 50%; text-align: center;">item code</td><td style="width: 50%; text-align: center;">buffer length</td></tr> <tr> <td colspan="2" style="text-align: center;">buffer address</td></tr> <tr> <td colspan="2" style="text-align: center;">return length address</td></tr> </table>		item code	buffer length	buffer address		return length address	
item code	buffer length						
buffer address							
return length address							

ZK-1705-84

The first field is a word containing a user-supplied integer specifying the length (in bytes) of the buffer in which the service writes the information. The length of the buffer needed depends upon the item code specified in the item code field of the item descriptor. If the value of buffer length is too small, the service truncates the data.

The second field is a word containing a user-supplied symbolic code specifying the item of information that the service is to return. These codes are defined by macros that are specific to the service.

The third field is a longword containing the user-supplied address of the buffer in which the service writes the information.

The fourth field is a longword containing the user-supplied address of a word in which the service writes the length in bytes of the information it actually returned.

item_quota_list

Structure that consists of one or more quota descriptors and that is terminated by a byte containing a value defined by the symbolic name PQL\$_LISTEND. Each quota descriptor consists of a 1-byte quota name followed by an unsigned longword containing the value for that quota.

lock_id

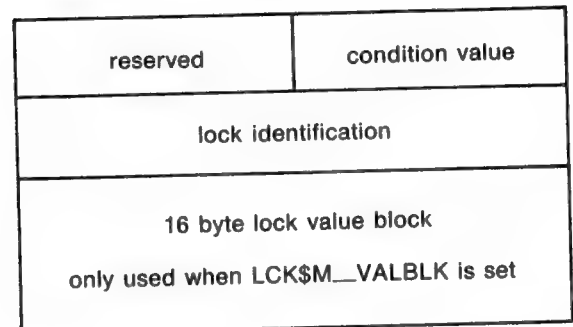
Unsigned longword integer denoting a lock identifier. This lock identifier is assigned by the lock manager facility to a lock when the lock is granted.

Introduction to System Services

Documentation Format for System Service Routines

Table 1-2 (Cont.) VMS Data Structures

Data Structure	Definition
lock_status_block	<p>Structure into which the lock manager facility writes status information about a lock. A lock status block always contains at least two longwords: the first word of the first longword contains a condition value; the second word of the first longword is reserved to DIGITAL; and the second longword contains the lock identifier.</p> <p>The lock status block receives the final condition value and the lock identification, and optionally contains a lock value block. When a request is queued, the lock identification is stored in the lock status block even if the lock has not been granted. This allows a procedure to dequeue locks that have not been granted.</p> <p>The condition value is placed in the lock status block only when the lock is granted (or when errors occur in granting the lock).</p> <p>The following diagram depicts a lock status block that includes the optional 16-byte lock value block.</p>



ZK-376-81

lock_value_block

16-byte block that the lock manager facility includes in a lock status block if the user requests it. The contents of the lock value block are user-defined and are not interpreted by the lock manager facility.

logical_name

Character string of from 1 to 255 characters that identifies a logical name or equivalence name to be manipulated by VMS logical name system services. Logical names that denote specific VMS objects have their own VMS types: for example, a logical name identifying a device has the VMS type "device_name".

longword_signed

This VMS data type is the same as the data type "longword integer (signed)" in Table 1-3.

Introduction to System Services

Documentation Format for System Service Routines

Table 1-2 (Cont.) VMS Data Structures

Data Structure	Definition
longword_unsigned	This VMS data type is the same as the data type "longword (unsigned)" in Table 1-3.
mask_byte	Unsigned byte wherein each bit is interpreted by the called procedure. A mask is also referred to as a set of "flags" or as a "bitmask".
mask_longword	Unsigned longword wherein each bit is interpreted by the called procedure. A mask is also referred to as a set of "flags" or as a "bitmask".
mask_quadword	Unsigned quadword wherein each bit is interpreted by the called procedure. A mask is also referred to as a set of "flags" or as a "bitmask".
mask_word	Unsigned word wherein each bit is interpreted by the called procedure. A mask is also referred to as a set of "flags" or as a "bitmask".
null_arg	Unsigned longword denoting a "null argument." A "null argument" is an argument whose only purpose is to hold a place in the argument list.
octaword_signed	This VMS data type is the same as the data type "octaword integer (signed)" in Table 1-3.
octaword_unsigned	This VMS data type is the same as the data type "octaword (unsigned)" in Table 1-3.
page_protection	Unsigned longword specifying page protection to be applied by the VAX hardware. Protection values are specified using bits 0 to 3; bits 4 to 31 are ignored.

Introduction to System Services

Documentation Format for System Service Routines

Table 1-2 (Cont.) VMS Data Structures

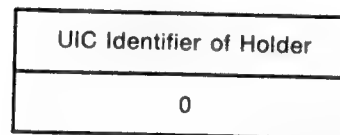
Data Structure	Definition
	The \$PRTDEF macro defines the following symbolic names for the protection codes:
Symbol	Description
PRT\$C_NA	No access
PRT\$C_KR	Kernel read only
PRT\$C_KW	Kernel write
PRT\$C_ER	Executive read only
PRT\$C_EW	Executive write
PRT\$C_SR	Supervisor read only
PRT\$C_SW	Supervisor write
PRT\$C_UR	User read only
PRT\$C_UW	User write
PRT\$C_ERKW	Executive read; kernel write
PRT\$C_SRKW	Supervisor read; kernel write
PRT\$C_SREW	Supervisor read; executive write
PRT\$C_URKW	User read; kernel write
PRT\$C_UREW	User read; executive write
PRT\$C_URSW	User read; supervisor write
procedure	<p>If the protection is specified as 0, the protection defaults to kernel read-only.</p> <p>Unsigned longword denoting the entry mask to a procedure that is not to be called at AST level. (Arguments specifying procedures to be called at AST level have the VMS type "ast_procedure".)</p>
process_id	Unsigned longword integer denoting a process identifier (PID). This process identifier is assigned by VMS to a process when the process is created.
process_name	Character string, containing 1 to 15 characters, that specifies the name of a process.
quadword_signed	This VMS data type is the same as the data type "quadword integer (signed)" in Table 1-3.
quadword_unsigned	This VMS data type is the same as the data type "quadword (unsigned)" in Table 1-3.

Introduction to System Services

Documentation Format for System Service Routines

Table 1-2 (Cont.) VMS Data Structures

Data Structure	Definition
rights_holder	Unsigned quadword specifying a user's access rights to a system object. This quadword consists of two fields: the first is an unsigned longword identifier (VMS type "rights_id") and the second is a longword bitmask wherein each bit specifies an access right. The following diagram depicts the format of a rights holder.



ZK-1903-84

rights_id

Unsigned longword denoting a rights identifier, which identifies an interest group in the context of the VMS security environment. This rights environment may consist of all or part of a user's User Identification Code (UIC).

Identifiers have two formats in the rights database: UIC format (VMS type "uic") and ID format. The high order bits of the identifier value specify the format of the identifier. Two high order zero bits identify a UIC format identifier; bit 31, set to one, identifies an ID format identifier.

Bit 31, set to one, specifies ID format. Bits 30 through 28 are reserved by DIGITAL. The remaining bits specify the identifier value. The following diagram depicts the ID format of a rights identifier.



ID Format

ZK-1906-84

To the system, an identifier is a binary value; however, to make identifiers easy to use, the system translates the binary identifier value into an identifier name. The binary value and the identifier name are associated in the rights database.

Introduction to System Services

Documentation Format for System Service Routines

Table 1-2 (Cont.) VMS Data Structures

Data Structure	Definition
	An identifier name consists of 1 to 31 alphanumeric characters and contains at least one nonnumeric character. An identifier name cannot consist entirely of numeric characters. It can include the characters A through Z, dollar signs (\$) and underscores (_), as well as the numbers 0 through 9. Any lowercase characters are automatically converted to uppercase.
rab	Structure denoting an RMS record access block. A complete description of this structure is contained in the <i>VAX Record Management Services Reference Manual</i> .
section_id	Unsigned quadword denoting a global section identifier. This identifier specifies the version of a global section and the criteria to be used in matching that global section.
section_name	Character string denoting a 1 to 43-character global-section name. This character string can be a logical name, but it must translate to a valid global-section name. For more information on how the system translates logical names to global section names see the "Memory Management" section of the <i>VAX/VMS System Services Reference Manual</i> .
system_access_id	Unsigned quadword that denotes a system identification value that is to be associated with a rights database.
time_name	Character string specifying a time value in VMS format.
uic	Unsigned longword denoting a User Identification Code (UIC). Each UIC is unique and represents a system user. The UIC identifier contains two high order bits that designate format, a member field, and a group field. Member numbers range from 0 to 65,534; group numbers range from 1 to 16,382. The following diagram depicts the UIC format.



UIC Format

ZK-1905-84

Introduction to System Services

Documentation Format for System Service Routines

Table 1-2 (Cont.) VMS Data Structures

Data Structure	Definition
user_arg	Unsigned longword denoting a user-defined argument. This longword is passed to a procedure as an argument, but the contents of the longword are defined and interpreted by the user.
varying_arg	Unsigned longword denoting a variable argument. A variable argument can have variable types, depending on specifications made for other arguments in the call.
vector_byte_signed	A homogeneous array whose elements are all signed bytes.
vector_byte_unsigned	A homogeneous array whose elements are all unsigned bytes.
vector_longword_signed	A homogeneous array whose elements are all signed longwords.
vector_longword_unsigned	A homogeneous array whose elements are all unsigned longwords.
vector_quadword_signed	A homogeneous array whose elements are all signed quadwords.
vector_quadword_unsigned	A homogeneous array whose elements are all unsigned quadwords.
vector_word_signed	A homogeneous array whose elements are all signed words.
vector_word_unsigned	A homogeneous array whose elements are all unsigned words.
word_signed	This VMS data type is the same as the data type "word integer (signed)" in Table 1-3.
word_unsigned	This VMS data type is the same as the data type "word (unsigned)" in Table 1-3.

1.1.3.2 Type Entry

When a calling program passes an argument to a system service, the service expects the argument to be of a particular data type. The service descriptions in Part II indicate the expected data types for each argument.

Properly speaking, an argument does not have a data type; rather, the data specified by an argument has a data type. The argument is merely the vehicle for the passing of data to the called routine.

As described in the VAX Procedure Calling Standard in *Introduction to VAX/VMS System Routines*, procedure calls result in the construction of an *argument list*. This argument list is a vector of longwords. The first longword on the list contains a count of the number of remaining longwords, and each remaining longword is one argument. Thus, an *argument* is one longword in the argument list.

Nevertheless, the phrase "argument data type" is frequently used to describe the data type of the data that is specified by the argument. This terminology is used because it is simpler and more straightforward than the strictly accurate phrase "data type of the data specified by the argument".

Introduction to System Services

Documentation Format for System Service Routines

The following table contains the data types allowed by the VAX Procedure Calling Standard.

Table 1-3 VAX Standard Data Types

Data Type	Symbolic Code
Absolute date and time	DSC\$K_DTYPE_ADT
Byte integer (signed)	DSC\$K_DTYPE_B
Bound label value	DSC\$K_DTYPE_BLV
Bound procedure value	DSC\$K_DTYPE_BPV
Byte (unsigned)	DSC\$K_DTYPE_BU
COBOL intermediate temporary	DSC\$K_DTYPE_CIT
D_floating	DSC\$K_DTYPE_D
D_floating complex	DSC\$K_DTYPE_DC
Descriptor	DSC\$K_DTYPE_DSC
F_floating	DSC\$K_DTYPE_F
F_floating complex	DSC\$K_DTYPE_FC
G_floating	DSC\$K_DTYPE_G
G_floating complex	DSC\$K_DTYPE_GC
H_floating	DSC\$K_DTYPE_H
H_floating complex	DSC\$K_DTYPE_HC
Longword integer (signed)	DSC\$K_DTYPE_L
Longword (unsigned)	DSC\$K_DTYPE_LU
Numeric string, left separate sign	DSC\$K_DTYPE_NL
Numeric string, left overpunched sign	DSC\$K_DTYPE_NLO
Numeric string, right separate sign	DSC\$K_DTYPE_NR
Numeric string, right overpunched sign	DSC\$K_DTYPE_NRO
Numeric string, unsigned	DSC\$K_DTYPE_NU
Numeric string, zoned sign	DSC\$K_DTYPE_NZ
Octaword integer (signed)	DSC\$K_DTYPE_O
Octaword (unsigned)	DSC\$K_DTYPE_OU
Packed decimal string	DSC\$K_DTYPE_P
Quadword integer (signed)	DSC\$K_DTYPE_Q
Quadword (unsigned)	DSC\$K_DTYPE_QU
Character string	DSC\$K_DTYPE_T
Aligned bit string	DSC\$K_DTYPE_V
Varying character string	DSC\$K_DTYPE_VT
Unaligned bit string	DSC\$K_DTYPE_VU
Word integer (signed)	DSC\$K_DTYPE_W
Word (unsigned)	DSC\$K_DTYPE_WU
Unspecified	DSC\$K_DTYPE_Z
Procedure entry mask	DSC\$K_DTYPE_ZEM
Sequence of instruction	DSC\$K_DTYPE_ZI

Introduction to System Services

Documentation Format for System Service Routines

1.1.3.3 Access Entry

The argument-access entry describes the way in which the called routine accesses the data specified by the argument. The following three methods of access are the most common.

- 1 Read only. Data upon which a routine operates, or data needed by the routine to perform its operation, must be *read* by the called routine. Such data is also called *input* data. When an argument specifies input data, the "access" entry shows "read only".

The term "only" is present to indicate that the called routine does not both read and write (that is, "modify") the input data. Thus, input data supplied by a variable is preserved when the called routine completes execution.

- 2 Write only. Data that the called routine returns to the calling routine must be *written* into a location where the calling routine can access it. Such data is also called *output* data. When an argument specifies output data, the "access" entry shows "write only".

The term "only" is present to indicate that the called routine does not read the contents of the location either before or after it writes into the location.

- 3 Modify. When an argument specifies data that is both read and written by the called routine, the "access" entry shows "modify". In this case, the called routine reads the input data, which it uses in its operation, and then overwrites the input data with the results (the output data) of the operation. Thus, when the called routine completes execution, the input data specified by the argument is lost.

The following is a complete list of the access types allowed by the VAX Procedure Calling Standard.

- Read only
- Write only
- Modify
- Function call (before return)
- JMP after unwind
- Call after stack unwind
- Call without stack unwind

1.1.3.4 Mechanism Entry

The way in which an argument specifies the actual data to be used by the called routine is defined in terms of the argument passing mechanism. There are three types of passing mechanism.

- 1 By value. When the longword argument in the argument list contains the actual data to be used by the routine, the actual data is said to be passed to the routine "by value". In this case, the longword argument contains the actual data; in other words, the argument is the actual data. Note that since an argument is only one longword in length, only data that can be represented in one longword can be passed by value.
- 2 By reference. When the longword argument in the argument list contains the address of the data to be used by the routine, the data is said to be passed "by reference". In this case, the argument is a pointer to the data.

Introduction to System Services

Documentation Format for System Service Routines

- 3** By descriptor. When the longword argument in the argument list contains the address of a descriptor, the data is said to be passed "by descriptor". A descriptor consists of two or more longwords (depending on the type of descriptor used), which describe the location, length, and data type of the data to be used by the called routine. In this case, the argument is a pointer to a descriptor that itself is a pointer to the actual data.

The following list contains the passing mechanisms allowed by the VAX Procedure Calling Standard.

Passing Mechanism	Descriptor Code
By value	
By reference	
By reference, array reference	
By descriptor	
By descriptor, fixed-length	DSC\$K_CLASS_S
By descriptor, dynamic string	DSC\$K_CLASS_D
By descriptor, array	DSC\$K_CLASS_A
By descriptor, procedure	DSC\$K_CLASS_P
By descriptor, decimal string	DSC\$K_CLASS_SD
By descriptor, noncontiguous array	DSC\$K_CLASS_NCA
By descriptor, varying string	DSC\$K_CLASS_VS
By descriptor, varying string array	DSC\$K_CLASS_VSA
By descriptor, unaligned bit string	DSC\$K_CLASS_UBS
By descriptor, unaligned bit array	DSC\$K_CLASS_UBA
By descriptor, string with bounds	DSC\$K_CLASS_SB
By descriptor, unaligned bit string with bounds	DSC\$K_CLASS_UBSB

1.1.3.5 Explanatory Text Entry

For each argument, one or more paragraphs of explanatory text follows the "type", "access", and "mechanism" entries. The first paragraph is highly structured and always contains the following items of information.

- 1** An initial sentence fragment that describes: (1) the nature of the data specified by the argument and (2) the way in which the routine uses this data. For example, if an argument were supplying a number, which the routine was to convert to another data type, the initial sentence fragment would be something like the following: "number that is to be converted to the such-and-such data type."
- 2** A sentence expressing the relationship between the argument and the data that it specifies. This relationship is the passing mechanism used to pass the data.

If the passing mechanism is "by value", this sentence says something like the following: "the xxx argument contains (or is) the such-and-such data."

If the passing mechanism is "by reference", this sentence says something like the following: "the xxx argument is the address of the such-and-such data."

Introduction to System Services

Documentation Format for System Service Routines

If the passing mechanism is "by descriptor", this sentence says something like the following: "the xxx argument is the address of a descriptor pointing to the such-and-such data."

Additional explanatory paragraphs appear for each argument as needed. For example, some arguments specify complex data consisting of many discrete fields, each of which has a particular purpose and use. In such cases, additional paragraphs provide detailed descriptions of each such field, symbolic names for the fields, if any, and guidance relating to their use.

1.1.4 Condition Values Returned Heading

A condition value is an unsigned longword that has several uses in the VAX architecture.

- It indicates the success or failure of a called procedure.
- It describes an exception condition when an exception is signaled.
- It identifies system messages.
- It reports program success or failure to the command language level.

Figure 2-2 in the *Introduction to VAX/VMS System Routines* depicts the format and contents of the longword condition value, and Section 1.2.5 in the *Introduction to VAX/VMS System Routines* describes these contents and explains in detail the uses of the condition value.

Under the *Condition Values Returned* heading, a two-column list gives the symbolic code for each condition value that the routine can return and its accompanying description. This description explains whether the condition value indicates success or failure, and if failure, what user action may have caused the failure and what can be done to correct it.

Symbolic codes for condition values are system defined. The symbolic code defined for each condition value equates to a number that is identical to the longword condition value when interpreted as a number. In other words, though the condition value consists of several fields, each of which can be interpreted individually for specific information, the entire longword condition value itself can be interpreted as an unsigned longword integer, and this integer has an equivalent symbolic code.

Note that if a called routine generates an exception condition during execution, the exception condition is *signaled*; the exception condition is then *handled* by a condition handler (either user-supplied or system-supplied). Depending on the nature of the exception condition and on the condition handler that handles the exception condition, the called routine will either continue normal execution or terminate abnormally.

The documentation heading *Condition Values Returned* describes the condition values returned by the routine when it completes execution without generating an exception condition.

Introduction to System Services

Documentation Format for System Service Routines

1.1.5 Condition Values Returned in the I/O Status Block Heading

When the called routine returns a condition value in an I/O status block, the possible condition values that the routine can return are listed under the "Condition Values Returned in the I/O Status Block" heading.

The routines that return condition values in the I/O status block are the system services that complete asynchronously.

Some system services complete asynchronously; that is, they return to the caller immediately after the call to the service is successfully queued but before the operation to be performed by the service has completed. This allows the calling program to continue execution while the system service itself is executing. System services that complete asynchronously all have arguments that specify an I/O status block. When the system service operation has completed, a condition value specifying the completion status of the operation is written to the I/O status block.

The first word in the I/O status block receives the condition value for the final completion status of an asynchronous system service. Representing a longword condition value in a word-length field is possible for system services because the high-order word in system service condition values is zero.

Section 1.2.5 in the *Introduction to VAX/VMS System Routines* explains the contents of the fields in the longword condition value in detail. However, the reason why a system service condition value can be represented in one word, rather than one longword, is worth mentioning here.

One field in the condition value specifies which facility generated the condition value; this field is in the high-order word of the longword condition value. For the system facility, the value of this field is zero. This fact allows condition values generated by the system facility (which includes all system services) to be represented in a word, rather than a longword, since bits in the high-order word are all zeros.



2

Calling System Services

System service procedures are called using the standard VAX procedure calling conventions. The programming languages that generate VAX native mode instructions provide mechanisms for specifying the procedure calls. These languages and supporting documentation are listed in the preface.

When you code a system service call, you must supply whatever arguments the service requires.

When the service completes execution, it returns control to the calling program with a return condition value. The caller should analyze the condition value to determine the success or failure of the service call, so the program can alter the flow of execution, if necessary.

If you are a VAX MACRO programmer, you should read Section 2.4 for details on how to write the instructions that generate system service calls.

If you program in either VAX MACRO or a high-level language, you should read Sections 2.2, 2.6, and 2.8. Section 2.2 provides information on specifying arguments to system services; Section 2.6 discusses methods for checking return status from system services. Section 2.8 provides programming examples in a number of VAX native languages to aid high-level language programmers in interpreting the programming examples that appear throughout Sections 3 through 13.

If you program in a high-level language, you should read Section 2.7 for information on how to call system services from high-level languages. For detailed information and examples, see the user's guide for your language.

System service macros generate argument lists and CALL instructions to call system services. These macros are located in the system library `SYS$LIBRARY:STARLET.MLB`. This library is searched automatically for unresolved references when you assemble a source program.

Knowledge of VAX MACRO rules for assembly language programming is required for understanding the material presented in this section. The *VAX MACRO Language Reference Manual* and the *VAX MACRO User's Guide* contain the necessary prerequisite information.

2.1

System Services and System Integrity

Many system services are available and suitable for application programs, but the use of some services must be restricted to protect the performance of the system and the integrity of user processes.

For example, because the creation of permanent mailboxes uses system dynamic memory, the unrestricted use of permanent mailboxes could decrease the amount of memory available to other users. Therefore, the ability to create permanent mailboxes is controlled: a user must be specifically assigned the privilege to use the Create Mailbox (`$CREMBX`) system service to create a permanent mailbox.

Calling System Services

System Services and System Integrity

The various controls and restrictions applied to system service usage are described below. The "Description" section of each system service in Part II lists any privileges and quotas that are necessary to use the service.

2.1.1 User Privileges

The system manager, who maintains the user authorization file for the system, grants privileges to use protected system services. The user authorization file contains, in addition to profile information on each user, a list of specific user privileges and resource quotas.

When you log in to the system, the privileges and quotas you have been assigned are associated with the process created on your behalf. These privileges and quotas are applied to every image that the process executes.

When an image issues a call to a system service that is protected by privilege, the privilege list is checked. If you have been granted the specific privilege required, the image is allowed to execute the system service; otherwise, a condition value indicating an error is returned.

For a list of privileges see the Part II description of the Create Process (\$CREPRC) system service.

2.1.2 Resource Quotas

Many system services require certain system resources for execution. These resources include system dynamic memory and process quotas for I/O operations. When a system service is called that uses a resource controlled by a quota, the process's quota for that resource is checked. If the process has exceeded its quota, or if it has no quota allotment, an error condition value may be returned. Normally, when a system service is called and a required resource is not available, the process is placed in a wait state until the resource becomes available. Then, the service completes execution. This mode is called resource wait mode.

In a real-time environment, however, it may not be practical or desirable for a program to wait. In these cases, you can choose to disable resource wait mode, so that when a required resource is unavailable, control returns immediately to the calling program with an error condition value. You can disable (and reenable) resource wait mode with the Set Resource Wait Mode (\$SETRWM) system service.

How a program responds to the unavailability of a resource depends very much on the application and the particular service that is being called. In some instances, the program may be able to continue execution and retry the service call later. In other instances, it may be necessary only to note that the program is being required to wait.

2.1.3 Access Modes

A process can execute at any one of four access modes: user, supervisor, executive, or kernel. The access modes determine a process's ability to access pages of virtual memory. Each page has a protection code associated with it, specifying the type of access—read, write, or no access—allowed for each mode. The *VAX Architecture Handbook* provides additional information on access modes.

For the most part, user-written programs execute in user mode; system programs executing at the user's request (system services, for example) may execute at one of the other three, more privileged, access modes.

In some system service calls, the access mode of the caller is checked. For example, when a process tries to cancel timer requests, it can cancel only those requests that were issued from the same or less privileged access modes. For example, a process executing in user mode cannot cancel a timer request made from supervisor, executive, or kernel mode, because they are more privileged access modes.

Note that many system services use access modes to protect system resources, and thus employ a special convention for interpreting access mode arguments. You can specify an access mode using a numeric value or a symbolic name. Shown below are the access modes, their numeric values, and symbolic names.

Access Mode	Numeric Value	Symbolic Name	Privilege Rank
Kernel	0	PSL\$C_KERNEL	High
Executive	1	PSL\$C_EXEC	.
Supervisor	2	PSL\$C_SUPER	.
User	3	PSL\$C_USER	Low

The symbolic names are defined by the symbolic definition macro \$PSLDEF.

System services that permit an access mode argument will only allow callers to specify an access mode less privileged than, or equal in privilege to, the access mode from which the service was called. If the access mode specified is more privileged than the access mode from which the service was called, the less privileged access mode is used.

To determine the mode to use, VAX/VMS compares the specified access mode with the access mode from which the service was called. If the modes are different, the less privileged access mode is always used. Because this operation results in an access mode with a higher numeric value (when the access mode of the caller is different from the specified access mode), the access mode is said to be "maximized."

Since much of the code you write will execute in user mode, you can omit the access mode argument. The argument value defaults to 0 (kernel mode), and when this value is compared with the value of the current execution mode (3, user mode), the higher value (3) is used.

Calling System Services

Determining Arguments for System Services

2.2 Determining Arguments for System Services

You can determine the arguments required by a system service from the service's descriptions in Part II. The "Format" section in each system service description indicates the positional dependencies and keyword names of each argument, as shown in the following sample:

`$SERVICE arga ,argb ,argc ,argd`

This format indicates that the macro name of the service is `$SERVICE` and that it requires four arguments, ordered as shown and with keyword names `arga`, `argb`, `argc`, and `argd`. The following format must be used for the argument list for this service.

31	8	7	0
0		4	
arga			
argb			
argc			
argd			

ZK-854-82

All arguments are longwords. The first longword in the list must always contain, in its low-order byte, the number of arguments in the remainder of the list. The remaining three bytes must be zeros.

Many arguments to system services are optional; these are indicated in the macro formats by brackets. For example, if the second and third arguments of `$SERVICE` are optional, the macro format would appear as follows:

`$SERVICE arga ,[argb] ,[argc] ,argd`

If you omit an optional argument in a system service macro, the macro supplies a default value for the argument.

Arguments that are optional to system services always have default values, regardless of whether they are passed by value, by reference or by descriptor. In almost every case, an optional argument defaults to 0. The macros used to call the system services allow some languages to set default values to values other than 0 (VAX MACRO and VAX BLISS-32 allow this). Default values other than 0 are specified in the description of the argument in Part II.

The description of an optional argument always specifies what action the service takes when the default value is used.

Arguments that specify a return address may be optional when the system service returns information; if the program does not require the information, you can omit the optional argument.

2.3 Obtaining Values for Symbolic Codes

Individual services have symbolic codes for special return conditions, argument list offsets, identifiers, and flags associated with these services. For example, the Create Process (\$CREPRC) service (which is used to create a subprocess or a detached process) has symbolic codes associated with the various privileges and quotas you can grant to the created process.

The default system macro library, STARLET.MLB, contains the macro definitions for most system symbols. When you assemble a source program that calls any of these macros, the assembler automatically searches STARLET.MLB for the macro definitions. Each symbol name has a numeric value.

If your language has a method of obtaining values for these symbols, this method is explained in the user's guide.

If your language does not have such a method, you can do the following:

- Write a short VAX MACRO program containing the desired macro(s).
- Assemble the program and generate a listing. Using the listing, find the desired symbols and their hexadecimal values.
- Define each symbol with its value within your source program.

For example, to use the Get Job/Process Information (\$GETJPI) service to find out the accumulated CPU time (in 10-millisecond ticks) for a specified process, you must obtain the value associated with the item identifier JPI\$_CPUTIM. You can do this in the following way:

- Create the following three-line VAX MACRO program (named JPIDEF.MAR here; you may choose any name you wish).

```
.TITLE JPIDEF Obtain values for $JPIDEF
$JPIDEF GLOBAL          ; these MUST be UPPERCASE
.END
```

- Assemble and link the program to create the file, JPIDEF.MAP.

```
$ MACRO JPIDEF
$ LINK/NOEXE/MAP/FULL JPIDEF
%LINK-W-USRTFR, image ML:[] .EXE; has no user transfer address
```

The file JPIDEF.MAP will contain the symbols defined by \$JPIDEF listed both alphabetically and numerically.

- Find the value of JPI\$_CPUTIM and define the symbol in your program.

2.4 Calling System Services from VAX MACRO

System service macros generate argument lists and CALL instructions to call system services. These macros are located in the system library SYS\$LIBRARY:STARLET.MLB. This library is searched automatically for unresolved references when you assemble a source program.

Knowledge of VAX MACRO rules for assembly language programming is required for understanding the material presented in this section. The *VAX MACRO Language Reference Manual* and the *VAX MACRO User's Guide* contain the necessary prerequisite information.

Calling System Services

Calling System Services from VAX MACRO

Each system service has four macros associated with it. These macros allow you to define symbolic names for argument offsets, construct argument lists for system services, and call system services. The generic macros and the functions they serve are described in the following list.

Macro	Function
\$nameDEF	Defines symbolic names for the argument list offsets
\$name	Defines symbolic names for the argument list offsets and constructs the argument list
\$name__S	Calls system service and constructs the argument list
\$name__G	Calls system service and uses argument list constructed by \$name macro

2.4.1 Using Macros to Construct Argument Lists

There are two generic macros for constructing argument lists for system services.

\$name

\$name__S

The macro you use depends on which macro you are going to use to call the system service. If you use the \$name__G macro to call a system service, you should use the \$name macro to construct the argument list. If you use the \$name__S macro to call a system service, you can also use it to construct the argument list.

2.4.1.1 Specifying Arguments with the \$name__S Macro and the \$name Macro

When you use the \$name__S or the \$name macro to construct an argument list for a system service, you can specify arguments in any of three ways:

- By using keywords to describe the arguments. All keywords must be followed by an equal sign (=) and then by the value of the argument.
- By using positional order, with omitted arguments indicated by commas in the argument positions. You can omit commas for optional trailing arguments.
- By using both positional dependence and keyword names (positional arguments must be listed first).

For example, \$SERVICE might have the following format:

\$SERVICE *arga* , [*argb*] , [*argc*] , *argd*

Assume, for the purposes of this example, that *arga* and *argb* are arguments that require you to specify numeric values and that *argc* and *argd* require you to specify addresses.

The following two examples show valid ways of writing the \$name__S macro to call \$SERVICE.

Calling System Services

Calling System Services from VAX MACRO

\$name_S Example 1: Using Keywords

```
MYARGD: .LONG 100
```

```
.$SERVICE_S ARGB=#0,ARGC=0,ARGA=#1,ARGD=MYARGD
```

\$name_S Example 2: Specifying Arguments in Positional Order

```
MYARGD: .LONG 100
```

```
.$SERVICE_S #1,,,MYARGD
```

The argument list is pushed on the stack as follows:

```
PUSHAL MYARGD
PUSHL  #0
PUSHL  #0
PUSHL  #1
```

Note that all arguments, whether specified positionally or with keywords, must be valid assembler expressions, since they are used as source operands in instructions.

The two following examples show valid ways of writing a \$name macro to construct an argument list for a later call to \$SERVICE.

\$name Example 1: Using Keywords

```
LIST:  $SERVICE -
        ARGB=0, -
        ARGC=0, -
        ARGA=1, -
        ARGD=MYARGD
```

\$name Example 2: Specifying Arguments in Positional Order

```
LIST:  $SERVICE -
        1,,,MYARGD
```

The argument list generated in both cases is:

```
LIST:  .LONG 4
        .LONG 1
        .LONG 0
        .LONG 0
        .ADDRESS -
        MYARGD
```

Note that all arguments, whether specified in positional order or by keyword, must be expressions that the assembler can evaluate to generate .LONG or .ADDRESS data directives. Contrast this with the arguments for the \$name_S macro, which must be valid assembler expressions, since they are used as source operands in instructions.

Calling System Services

Calling System Services from VAX MACRO

2.4.1.2 Conventions for Specifying Arguments to System Services

The arguments must be specified according to the VAX MACRO assembler rules for operand specifying and addressing.

The way to specify a particular argument depends on the following factors:

- Whether the system service requires an address or a value as the argument. In Part II, the descriptions of the arguments following a system service macro format always indicate if the argument is an address. A Boolean value, number, or mask takes a value as the argument.
- The system service macro being used. The expansions of the \$name and \$name__S macros in the examples in the preceding sections showed the code generated by each macro.

If you do not know whether you have specified a value or an address argument correctly, you can assemble the program with the .LIST MEB directive to check the macro expansion. See the *VAX MACRO Language Reference Manual* for details.

2.4.1.3 Defining Symbolic Names for Argument List Offsets: \$name and \$nameDEF

You can refer symbolically to arguments in the argument list. Each argument in an argument list has an offset from the beginning of the list; a symbolic name is defined for the numeric offset of each argument. If you use the symbolic names to refer to the arguments in a list, you do not have to remember the numeric offset (which is based on the position of the argument shown in the macro format).

There are two additional advantages to referring to arguments by their symbolic names:

- 1 Your program is more readable.
- 2 If an argument list for a system service changes with a later release of a system, the symbols will not change.

The offset names for all system service argument lists are formed by concatenating the service macro name with \$_ and the keyword name of the argument. Here name is the name for the system service macro and keyword is the keyword argument.

name\$_keyword

Similarly, the number of arguments required by a particular macro is defined symbolically as:

name\$_NARGS

Symbolic names for argument list offsets are defined automatically whenever you use the \$name macro for a particular system service. You can also define symbolic names for system service argument lists using the \$nameDEF macro. This macro does not generate any executable code; it merely defines the symbolic names so they can be used later in the program. For example:

\$QIODEF

This macro defines the symbol QIO\$_NARGS and the symbolic names for the \$QIO argument list offsets.

You may need to use the \$nameDEF macro if you specify an argument list to a system service without using the \$name macro; or if a program refers to an argument list in a separately assembled module.

Calling System Services

Calling System Services from VAX MACRO

For example, the \$READEF and \$READEFDEF macros define the values listed below.

Symbolic Name	Value
READEF\$_NARGS	Number of arguments in the list (2)
READEF\$_EFN	Offset of EFN argument (4)
READEF\$_STATE	Offset of STATE argument (8)

Thus, the \$READEF macro can be specified to build an argument list for a \$READEF system service call as follows:

```
READLST:  $READEF  EFN=1, STATE=TEST1
```

Later, the program may want to use a different value for the **state** argument in calling the service. The following lines show how, with a call to the \$name_G macro, this can be accomplished.

```
MOVAL  TEST2, READLST+READEF$_STATE
$READEF_G READLST
```

The MOVAL instruction replaces the address TEST1 in the \$READEF argument list with the address TEST2; the \$READEF_G macro calls the system service with the modified list.

2.4.2 Using Macros to Call System Services

There are two generic macros for writing calls to system services:

```
$name_S
$name_G
```

Which macro you use depends on how the argument list for the system service is constructed.

- The \$name_S macro requires you to supply the arguments to the system service in the system service macro. The macro generates code to push the argument list onto the call stack during program execution. With this macro, you can use registers to contain or point to arguments so that you can write reentrant programs.
- The \$name_G macro requires you to construct an argument list elsewhere in the program and specify the address of this list as an argument to the system service. (A macro is provided to create an argument list for each system service.) With this macro, you can use the same argument list, with modifications if necessary, for more than one invocation of the macro.

The \$name_S macro generates a CALLS instruction; the \$name_G macro generates a CALLG instruction. The services are called according to the standard procedure calling conventions. System services save all registers except R0 and R1, and restore the saved registers before returning control to the caller.

The sections which follow describe how to code system service calls using each of these macros.

Calling System Services

Calling System Services from VAX MACRO

2.4.2.1 The \$name_S Macro

The format of \$name_S macro call is:

\$name_S arg1, ..., argn

The macro generates code to push the arguments on the stack in reverse order. The actual instructions used to place the arguments on the stack are determined as follows:

- If the system service requires a value for an argument, either a PUSHL instruction or a MOVZWL to -(SP) instruction is generated.
- If the system service requires an address for an argument, a PUSHAB, PUSHAW, PUSHAL, or PUSHQAQ instruction is generated, depending on the context.

The macro then generates a call to the system service in the following format:

CALLS #n,@@SYS\$name

In this format, n is the number of arguments on the stack.

2.4.2.2 Example of \$name_S Macro Call

Since a \$name_S macro constructs the argument list at execution time, addresses and values can be supplied using register addressing modes. The following line can be used to execute the \$READEFS macro:

\$READEFS EFN=#1,STATE=(R10)

R10 contains the address of the longword to receive the status of the flags.

This macro instruction is expanded as follows:

```
PUSHAL (R10)
PUSHL #1
CALLS #2,@@SYS$READEF
```

2.4.2.3 The \$name_G Macro

The \$name_G macro requires a single operand:

\$name_G label

label

address of the argument list.

The \$name Macro

Macros are provided to create argument lists for the \$name_G macro. The format of the macros is as follows:

label: \$name arg1,...,argn

label

symbolic address of the generated argument list. This is the label given as an argument in the \$name_G macro.

\$name

the service macro name.

arg1,...,argn

arguments to be placed in successive longwords in the argument list.

Calling System Services

Calling System Services from VAX MACRO

The \$name_G macro (used with the \$name macro) is especially useful for doing the following:

- Making calls to system services that have long argument lists
- Calling services repeatedly during the execution of a single program with the same, or essentially the same, argument list

2.4.2.4 Example of \$NAME and \$name_G Macro Calls

This example shows how you can write a call to the Read Event Flags (\$READEF) system service using an argument list created by \$name.

As shown in Part II, the macro format of the \$READEF system service is:

```
$READEF efn ,state
```

The **efn** argument must specify the number of an event flag cluster, and the **state** argument must supply the address of a longword to receive the contents of the cluster.

These arguments might be specified using the \$name macro as follows:

```
READLST:
    $READEF EFN=1, -           ; argument list for $READEF
    STATE=TESTFLAG
```

This \$READEF macro generates the code:

```
READLST:
    .LONG 2           ; argument list for $READEF
    .LONG 1
    .ADDRESS -
    TESTFLAG
```

To execute the \$READEF macro now requires only the following line:

```
$READEF_G READLST
```

The macro generates the following code to call the Read Event Flags system service:

```
CALLG READLST,C#SYS$READEF
```

SY\$READEF is the name of a vector to the entry point of the Read Event Flags system service. The linker automatically resolves the entry point addresses for all system services.

2.5 System Service Completion

When a system service completes, control is returned to your program. You can choose how and when control is returned to your program by choosing synchronous or asynchronous forms of system services and by enabling process execution modes.

The sections that follow describe:

- When synchronous system services return control to your program
- When asynchronous system services return control to your program
- How you can synchronize the completion of asynchronous system services
- How control is returned to your program when special process execution modes are enabled

Calling System Services

System Service Completion

2.5.1 Synchronous and Asynchronous System Services

A number of system services can be executed either synchronously or asynchronously (for example, SYS\$GETJPI and SYS\$GETJPIW). The "W" at the end of the system service name indicates the synchronous version of the system service.

The asynchronous version of a system service queues a request and returns control to your program. You can perform operations while the system service executes; however, do not attempt to access information returned by the service until checking that the system service has completed.

Typically, you pass an asynchronous system service an event flag and an I/O status block. When the system service completes, it sets the event flag and places the final status of the request in the I/O status block. Use the SYS\$SYNCH system service to ensure that the system service has completed. You pass SYS\$SYNCH the event flag and I/O status block that you passed to the asynchronous system service; SYS\$SYNCH waits for the event flag to be set, then ensures that the system service rather than some other program set the event flag by checking the I/O status block. If the I/O status block is still zero, SYS\$SYNCH waits until the I/O status block is filled.

The synchronous version of a system service acts exactly as if you had used the asynchronous version followed immediately by a call to SYS\$SYNCH. Regardless of whether you use the synchronous or asynchronous version of a system service, if you omit the *efn* argument, the service uses event flag number zero.

Example of \$SYNCH System Service in VAX FORTRAN

```
! data structure for SYS$GETJPI
.
.
.
INTEGER*4 STATUS,
2      FLAG,
2      PID_VALUE
! I/O status block
INTEGER*2 JPISTATUS,
2      LEN
INTEGER*4 ZERO /0/
COMMON /IO_BLOCK/ JPISTATUS,
2      LEN,
2      ZERO
.
.
.
! Call SYS$GETJPI and wait for information
STATUS = LIB$GET_EF (FLAG)
IF (.NOT. STATUS) CALL LIB$SIGNAL (XVAL(STATUS))
STATUS = SYS$GETJPI (XVAL(FLAG),
2      PID_VALUE,
2      ,
2      NAME_BUF_LEN,
2      JPISTATUS,
2      ,)
IF (.NOT. STATUS) CALL LIB$SIGNAL (XVAL(STATUS))
.
.
.
STATUS = SYS$SYNCH (XVAL(FLAG),
2      JPISTATUS)
IF (.NOT. JPISTATUS) THEN
    CALL LIB$SIGNAL (XVAL(JPISTATUS))
END IF
END
```

2.5.2 Process Execution Modes

Two process execution modes affect how control is returned to the calling program when an error occurs during the execution of a system service. These modes are:

- Resource wait mode
- System service failure exception mode

If you change the default setting for either of these modes in a program, the program must handle the special return conditions that result. The next two sections discuss considerations for using these modes.

2.5.2.1 Resource Wait Mode

Normally, when a system service is called and a required resource is not available, the process is placed in a wait state until the resource becomes available. Then, the service completes execution. This mode is called resource wait mode.

In a real-time environment, however, it may not be practical or desirable for a program to wait. In these cases, you can choose to disable resource wait mode, so that when a required resource is unavailable, control returns immediately to the calling program with an error condition value. You can disable (and reenable) resource wait mode with the Set Resource Wait Mode (\$SETRWM) system service.

How a program responds to the unavailability of a resource depends very much on the application and the particular service that is being called. In some instances, the program may be able to continue execution and retry the service call later. In other instances, it may be necessary only to note that the program is being required to wait.

2.5.2.2 System Service Failure Exception Mode

When an error occurs during the execution of a system service, control normally returns to the next instruction in the calling program, which can check the return condition value in R0 to determine the success or failure of the service call.

To detect and respond to system service call failures, you can use the condition-handling mechanism of VAX/VMS to respond to system service failures. Then, when an error occurs, a software exception condition is generated, and control is passed to a condition-handling routine.

This mode is called system service failure exception mode, and can be enabled (and disabled) with the Set System Service Failure Exception Mode (\$SETSFME) system service, as shown in the following example.

```
#SETSFME ENBFLG=#1
```

This call enables the generation of exceptions when errors or severe errors occur during execution of a system service (exceptions are not generated for warning returns).

Certain formatting and conversion services are not affected by the enabling of system service failure exception mode. The following services will not generate exceptions when failures occur and system service failure exception mode is enabled.

Calling System Services

System Service Completion

```
$ASCTIM  
$BINTIN  
$FAO/$FAOL  
$PUTMSG  
$UNWIND
```

If you write a program to execute with this mode enabled, you can write a condition-handling routine. Information on condition handlers is provided in Section 10, Condition-Handling Services. If no user-specified routine is available when an exception occurs and the program was run with the DCL command RUN, the default condition handler causes the program to exit and displays descriptive information about the exception condition.

DIGITAL recommends that high-level language programs not enable system service failure exception mode, except perhaps in certain debugging situations. If you enable system service failure exception mode and do not declare your own condition handler, many error messages displayed at run time will be meaningless. High-level language compilers generate calls to system services for many statements or instructions in source programs. (For example, reads and writes to files generate calls to VAX RMS, which uses the \$QIO and \$QIOW services.) If you enable system service failure exception mode, many different types of errors—such as an I/O attempt to a nonexistent device or non-numeric input to a math routine—will generate the message “%SYSTEM-F-SSFAIL, system service failure exception,...”.

2.6 Condition Values Returned from System Services

When a system service finishes execution, a numeric status value is always returned. For VAX MACRO calls, the status value is returned in general register R0; the mechanisms used in high-level languages vary, see the user's guide for the appropriate language.

Depending on your specific needs, you can test just the low-order bit, the low-order three bits, or the entire value.

- The low-order bit indicates successful (1) or unsuccessful (0) completion of the service.
- The low-order three bits, taken together, represent the severity of the error. The severity code values are listed below.

Value	Meaning	Symbolic Name
0	Warning	STS\$K_WARNING
1	Success	STS\$K_SUCCESS
2	Error	STS\$K_ERROR
3	Informational	STS\$K_INFO
4	Severe or fatal error	STS\$K_SEVERR
5-7	Reserved	

The symbolic names are defined by the symbolic definition macro \$STSDEF.

- The remaining bits (bits 3 through 31) classify the particular return condition and the operating system component that issued the condition value. For system service return status values, the high-order word (bits 16 through 31) contains zeros.

Calling System Services

Condition Values Returned from System Services

Each numeric condition value has a unique symbolic name in the following format, where code is a mnemonic describing the return condition.

`SS$_code`

For example, a successful return is usually indicated by:

`SS$_NORMAL`

An example of an error return condition value is:

`SS$_ACCVIO`

This condition value indicates that an access violation occurred because a service could not read an input field or write an output field.

The symbolic definitions for condition values are included in the default system library (SYS\$LIBRARY:STARLET.OLB). You can obtain a listing of these symbolic codes at assembly time by invoking the system macro `SS$DEF`. Use the symbolic names for system condition values to check return conditions.

VAX/VMS does not automatically handle system service failure or warning conditions; you must test for them and handle them yourself. This contrasts with the operating system's handling of exception conditions detected by the hardware or software; the system handles these exceptions by default, although you can intervene in or override the default handling by declaring a condition handler (see Section 10, Condition-Handling Services).

2.6.1 Information Provided by Condition Values

Condition values returned by system services may provide information; that is, they do not always just indicate whether or not the service completed successfully. `SS$_NORMAL` is the usual condition value indicating success, but others are defined. For example, the condition value `SS$_BUFFEROVF`, which is returned when a character string returned by a service is longer than the buffer provided to receive it, is a success code. This condition value, however, gives the program additional information.

Warning returns and some error returns indicate that the service may have performed some part, but not all, of the requested function.

The possible condition values that each service can return are described with the individual service descriptions in Part II. When you write calls to system services, read the descriptions of the return condition values to determine whether you want the program to check for particular return conditions.

2.6.2 Testing Return Condition Values

To test for successful completion following a system service call, the program can test the low-order bit of `R0` and branch to an error checking routine if this bit is not set, as follows:

```
BLBC    R0,errorlabel      ; error if low bit clear
```

Programs should not test for success by comparing the return status to `SS$_NORMAL`. A future release of VAX/VMS might add new alternate success codes to an existing service, causing programs that test for `SS$_NORMAL` to fail.

Calling System Services

Condition Values Returned from System Services

The error checking routine may check for specific values or for specific severity levels. For example, the following instruction checks for an illegal event flag number error condition:

```
CNPL    $$$_ILLEFC,R0          ; is event flag number illegal?
```

Note that return condition values are always longword values; however, the high-order words of all condition values returned in R0 by system services are always the same.

2.6.3 System Messages Generated by Condition Values

When you execute a program with the DCL command RUN, the command interpreter uses the contents of R0 to issue a descriptive message if the program completes with a nonsuccessful status.

The following code fragment shows a simple error-checking procedure in a main program:

```
        $READEFS -  
            EFN=84, -  
            STATE=TEST  
        BSBW    ERROR  
        .  
        .  
ERROR:   BLBC    R0,10$          ; check register 0  
        RSB                      ; success, return  
10$:    RET                      ; exit with R0 status
```

Following a system service call, the BSBW instruction branches to the subroutine ERROR. The subroutine checks the low-order bit in register 0 and if the bit is clear, branches to a RET instruction that causes the program to exit with the status of R0 preserved. Otherwise, the subroutine issues an RSB instruction to return to the main program.

If the event flag cluster requested in this call to \$READEFS is not currently available to the process, the program exits and the command interpreter displays the following message:

```
%SYSTEM-F-UNASEFC, unassociated event flag cluster
```

The keyword UNASEFC in the message corresponds to the condition value \$\$\$_UNASEFC.

Note that three severe errors that are generated by the calls themselves, not the services, can be returned from calls to system services.

Error	Meaning
\$\$\$_ACCVIO	The argument list cannot be read by the caller (using the \$name_G macro), and the service is not called. This meaning of \$\$\$_ACCVIO is different from its meaning for individual services. When \$\$\$_ACCVIO is returned from individual services, the service is called, but one or more arguments to the service cannot be read or written by the caller.
\$\$\$_INSFARG	Not enough arguments were supplied to the service.
\$\$\$_ILLSER	An illegal system service was called.

2.7 High-Level Language Calls

Each high-level language supported by VAX/VMS provides some mechanism for calling an external procedure and passing arguments to that procedure. The specifics of the mechanism and the terminology used, however, vary from one language to another. It is not possible for this manual to describe the ways in which each high-level language calls system services. For specific information, DIGITAL recommends that you refer to the appropriate high-level language user's guide.

VAX/VMS system services are external procedures that accept arguments. There are three ways to pass arguments to system services.

- 1 By value. When the longword argument in the argument list contains the actual data to be used by the routine, the actual data is said to be passed to the routine by value. In this case, the longword argument contains the actual data; in other words, the argument is the actual data. Note that since an argument is only one longword in length, only data that can be represented in one longword can be passed by value.
- 2 By reference. When the longword argument in the argument list contains the address of the data to be used by the routine, the data is said to be passed by reference. In this case, the argument is a pointer to the data.
- 3 By descriptor. When the longword argument in the argument list contains the address of a descriptor, the data is said to be passed by descriptor. A descriptor consists of two or more longwords (depending on the type of descriptor used), which describe the location, length, and data type of the data to be used by the called routine. In this case, the argument is a pointer to a descriptor that itself is a pointer to the actual data.

The description of each service in Part II of this manual indicates how each argument is to be passed. Phrases such as "an address" and "address of a character string descriptor" identify reference and descriptor arguments, respectively. Words like "Boolean value," "number," "value," or "mask" indicate an argument passed by value. Figure 2-1 shows how arguments are passed to the system services.

Some services also require service-specific data structures that indicate functions to be performed or hold information to be returned. Descriptions of these service-specific data structures are included in the Part II reference section for the service. You can use this information and information from your programming language manuals to define such service-specific item lists.

2.7.1 Testing Return Condition Values in High-level Languages

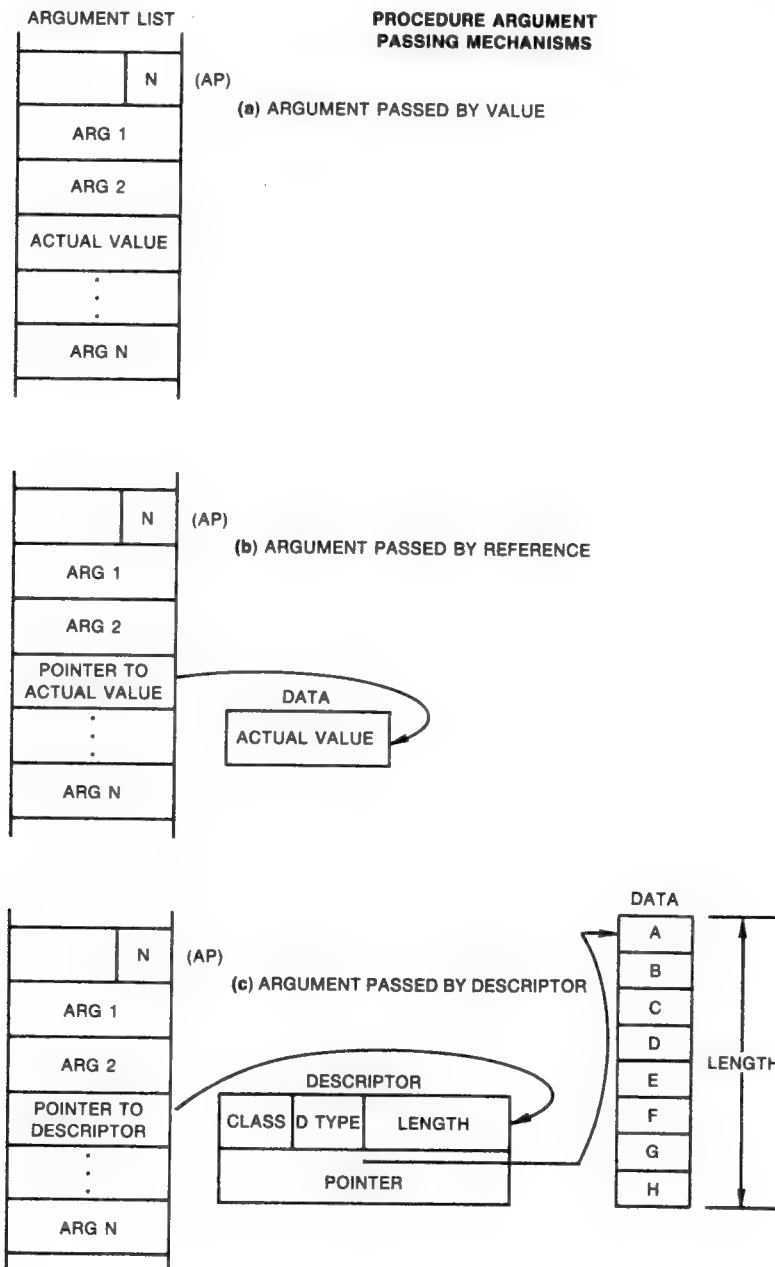
When a service returns control to your program, it places a return status value in the general register R0. The value in the low-order word indicates either that the service completed successfully or that some specific error prevented the service from performing some or all of its functions. After each call to a system service, you must check whether it completed successfully. You can also test for specific error conditions. (See Section 2.6 for more information on return status values.)

Each language provides some mechanism for testing the return status. Often you need only check the low-order bit, such as by a test for TRUE (success or informational return) or FALSE (error or warning return).

Calling System Services

High-Level Language Calls

Figure 2-1 Procedure Argument Passing Mechanisms



Note: ARG 1, ARG 2, ARG N can be passed by value, by reference, or by descriptor in any of the above examples.

:(AP) = argument pointer

N = number of arguments

ZK-1962-84

Calling System Services

High-Level Language Calls

To check the entire value for a specific return condition, each language provides a way for your program to determine the values associated with specific symbolically defined codes. You should always use these symbolic names when you write tests for specific conditions.

For information on how to test for these codes, see the user's guide for your programming language.

2.8 Interpreting the Programming Examples

Sections 3 through 13 contain many programming examples (using VAX MACRO and VAX FORTRAN) designed to familiarize you with the system services and their arguments. The examples do not show complete programming sequences; rather, they show only the code and/or arguments pertinent to a particular discussion.

In some of the more complex examples, explanatory text is keyed to the example using a special numeric symbol, for example.

Although the examples are written using VAX MACRO and VAX FORTRAN, they are designed to be as meaningful as possible to programmers using other high-level languages as well. Figure 2-2 provides additional help to high-level language programmers in interpreting the MACRO examples. This figure shows a portion of a VAX MACRO program and the "equivalent" in the following languages:

- VAX BASIC
- VAX BLISS-32
- VAX COBOL
- VAX FORTRAN
- VAX PASCAL

Calling System Services

Interpreting the Programming Examples

Figure 2-2 Interpreting MACRO Examples

MACRO Example

```

CYGDES: .ASCID /CYGNUS/ ② ; descriptor for CYGNUS string
TBLDES: .ASCID /LNN$SYSTEM/ ③ ; logical name table
NAMBUF: .BLKB 255 ④ ; output buffer
NAMLEN: .BLKW 1 ⑤ ; word to receive length
ITEMS: .WORD 255 ; output buffer length
        .WORD LNN$STRING ; item code
        .ADDRESS - ; output buffer
            NAMBUF
        .ADDRESS - ; return length
            NAMLEN
        .LONG 0 ; list terminator
        .
        .ENTRY ORION,0 ① ; routine entry point & mask
⑥ $TRNLNN_S -
            TABNAM=TBLDES, -
            LOGNAM=CYGDES, -
            ITMLST=ITEMS
⑦ BLBC RO,ERROR ; check for error
        .
        .END

```

(Continued on next page)

Calling System Services

Interpreting the Programming Examples

Figure 2-2 (Cont.) Interpreting MACRO Examples

MACRO Notes

- ❶ A routine name and entry mask show the beginning of executable code in a routine or subroutine.
- ❷ The input character string descriptor argument is defined using the .ASCID directive.
- ❸ The name of the table to search is defined using the .ASCID directive.
- ❹ For an output character string argument, allocate enough bytes to hold the output data.
- ❺ The MACRO directive .BLKW reserves a word to hold the output length.
- ❻ Call the service by a macro name that has the suffix _S or _G.

You can specify arguments by keyword (as in this example) or in positional order. (Keyword names correspond to the names of the arguments shown in lowercase in the system service format descriptions in Part II.) If you omit any optional arguments (that is, accept the defaults), you can omit them completely if you specify arguments by keyword, but you must specify the comma for each missing argument if you specify arguments by positional order.

Use the number sign (#) to indicate a literal value for an argument.

- ❶ The BLBC instruction causes a branch to a subroutine named ERROR (not shown) if the low bit of the condition value returned from the service is clear (low bit clear = failure or warning). You can use a BSBW instruction to branch unconditionally to a routine that checks the return status.

(Continued on next page)

Calling System Services

Interpreting the Programming Examples

Figure 2-2 (Cont.) Interpreting MACRO Examples

BASIC Equivalent

```

10 SUB ORION ①                                ! Subprogram ORION
    OPTION TYPE=EXPLICIT                      ! Require declaration of all
                                              ! symbols
    EXTERNAL LONG FUNCTION SYS$TRNLNM        ! Declare the system service
    EXTERNAL WORD CONSTANT LNM$STRING        ! The request code that
                                              ! we will use
    DECLARE WORD NAMLEN, ④                    ! Word to receive length
    LONG SYS_STATUS                          ! Longword to receive status
    COMMON (BUF) STRING NAME_STRING = 255 ⑤
    RECORD ITEM_LIST                        ! Define item
                                              ! descriptor structure
        WORD BUFFER_LENGTH                  ! The buffer length
        WORD ITEM                          ! The request code
        LONG BUFFER_ADDRESS                ! The buffer address
        LONG RETURN_LENGTH_ADDRESS          ! The address of the return len
                                              ! word
        LONG TERMINATOR                    ! The terminator
    END RECORD ITEM_LIST                    ! End of structure definition
    DECLARE ITEM_LIST ITEMS                 ! Declare an item list
    ITEMS::BUFFER_LENGTH = 255%             ! Initialize the item list
    ITEMS::ITEM = LNM$STRING
    ITEMS::BUFFER_ADDRESS = LOC( NAME_STRING )
    ITEMS::RETURN_LENGTH_ADDRESS = LOC( NAMLEN )
    ITEMS::TERMINATOR = 0

    SYS_STATUS = SYS$TRNLNM( , 'LNM$SYSTEM', 'CYGNUS',, ITEMS) ②
    IF (SYS_STATUS AND 1%) = 0% ⑥
    THEN
        ! Error path
    ELSE
        ! Success path
    END IF
END SUB

```

BASIC Notes

- ① The routine and its entry mask are defined by the SUB statement.
- ② Specify the input character string directly in the system service call; the compiler builds the descriptor.
- ③ The COMMON (BUF) STRING NAME_STRING = 255 declaration allocates 255 bytes for the output data in a static area. The compiler builds the descriptor.
- ④ The DECLARE WORD NAMLEN declaration reserves a 16-bit word for the output value.
- ⑤ Invoke the system service as a function using the SYS\$ form.
Enclose the arguments in parentheses, and specify them in positional order only. Specify a comma for each optional argument that you omit (including trailing arguments).
- ⑥ The IF statement performs a test on the low-order bit of the return status. This form is recommended for all status returns.

(Continued on next page)

Calling System Services

Interpreting the Programming Examples

Figure 2-2 (Cont.) Interpreting MACRO Examples

BLISS Equivalent

```
MODULE ORION=
BEGIN
EXTERNAL ROUTINE
  ERROR_PROC: NOVALUE;           ! Error processing routine
LIBRARY 'SYS$LIBRARY:STARLET.L32'; ! Library containing VAX/VMS
                                   ! macros (including $TRNLNM).
                                   ! This declaration
                                   ! is required.

GLOBAL ROUTINE ORION: NOVALUE=
  BEGIN
  OWN
    NAMBUF : VECTOR[255, BYTE],    ! Output buffer
    NAMLEN : WORD,                 ! Translated string length
    ITEMS : BLOCK[16, BYTE]
      INITIAL(WORD(255,           ! output buffer length
        LNM$STRING),             ! item code
        NAMBUF,                  ! output buffer
        NAMLEN,                  ! address of word for
                                ! translated
                                ! string length
                                ! list terminator
        0);
  LOCAL
    STATUS;                       ! Return status from
                                   ! system service

  STATUS = $TRNLNM(TABNAM = %ASCID'LNM$SYSTEM',
    LOGNAME = %ASCID'CYGNUS',
    ITMLST = ITEMS); ❶

  IF NOT .STATUS THEN ERROR_PROC(.STATUS); ❷
END;
```

BLISS Notes

- ❶ Invoke the macro by its service name, without a suffix.
Enclose the arguments in parentheses, and specify them by keyword. (Keyword names correspond to the names of the arguments shown in lowercase in the system service format descriptions in Part II.)
- ❷ The return status, which is assigned to the variable STATUS, is tested for TRUE or FALSE. FALSE (low bit = 0) indicates failure or warning.

(Continued on next page)

Calling System Services

Interpreting the Programming Examples

Figure 2-2 (Cont.) Interpreting MACRO Examples

COBOL Equivalent

```
IDENTIFICATION DIVISION.  
PROGRAM-ID. ORION. ①  
ENVIRONMENT DIVISION.  
DATA DIVISION.  
WORKING-STORAGE SECTION.  
01 TABNAM PIC X(11) VALUE "LNM$PROCESS".  
01 CYGDES PIC X(6) VALUE "CYGNUS".  
01 NAMDES PIC X(255) VALUE SPACES. ②  
01 NAMLEN PIC S9(4) COMP.  
01 ITMLIS.  
    02 BUFLEN PIC S9(4) COMP VALUE 225.  
    02 ITMCOB PIC S9(4) COMP VALUE 2. ③  
    02 BUFADR POINTER VALUE REFERENCE NAMDES.  
    02 RETLEN POINTER VALUE REFERENCE NAMLEN.  
    02 FILLER PIC S9(5) COMP VALUE 0.  
01 RESULT PIC S9(9) COMP. ④  
PROCEDURE DIVISION.  
START-ORION.  
    CALL "SYS$STRNLNM" ⑤  
    USING OMITTED  
        BY DESCRIPTOR TABNAM  
        BY DESCRIPTOR CYGDES ⑥  
        OMITTED  
        BY REFERENCE ITMLIS  
    GIVING RESULT.  
    IF RESULT IS FAILURE ⑦  
        GO TO ERROR-CHECK.  
    DISPLAY "NAMDES: ", NAMDES(1:NAMLEN).  
    GO TO THE-END.  
ERROR-CHECK.  
    DISPLAY "Returned Error: ", RESULT CONVERSION.  
THE-END.  
STOP RUN.
```

(Continued on next page)

Calling System Services

Interpreting the Programming Examples

Figure 2-2 (Cont.) Interpreting MACRO Examples

COBOL Notes

- ❶ The PROGRAM-ID paragraph identifies the program by specifying the program name, which is the global symbol associated with the entry point. The compiler builds the entry mask.
- ❷ Define the input string as alphanumeric (ASCII) data. The compiler generates a descriptor when you specify "USING BY DESCRIPTOR" in the CALL statement.
- ❸ Allocate enough bytes for the alphanumeric output data. The compiler generates a descriptor when you specify "USING BY DESCRIPTOR" in the CALL statement.
- ❹ This definition reserves a signed longword with COMP (binary) usage to receive the output value.
- ❺ Call the service using the "SYS\$" form of the service name, and enclose the name in quotation marks.

Specify arguments in positional order only, with "USING...". You cannot omit arguments; if you are accepting the default for an argument, you must explicitly pass the default value (OMITTED in this example).

You can specify explicitly how each argument is being passed: BY DESCRIPTOR, BY REFERENCE (that is, by address), or BY VALUE. You can also implicitly specify how an argument is being passed: through the default mechanism (BY REFERENCE), or through association with the last specified mechanism (thus, the last two arguments in the example are implicitly passed BY VALUE).

- ❻ The IF statement tests RESULT for a failure status. In this case, control is passed to the routine ERROR-CHECK.
- ❼ The value of the symbolic code LNM\$STRING is 2. Section 2.3 explains how to obtain values for symbolic codes.

(Continued on next page)

Calling System Services

Interpreting the Programming Examples

Figure 2-2 (Cont.) Interpreting MACRO Examples

FORTRAN Equivalent

```
SUBROUTINE ORION
C      Declare all the system service names, the output buffer,
C      and a variable to receive the length
C      of the string returned.
C
      INCLUDE '($SYSSRVNAM)' ①
      CHARACTER*255 EQUIV_NAME ②
      INTEGER*2 NAMLEN ③
      INTEGER*4 STATUS ④
      .
      .
      .
      ⑤ STATUS = SYS$STRNLN('CYGNUS', NAMLEN, EQUIV_NAME, XVAL(4))
      IF (.NOT. STATUS) CALL EXIT(STATUS) ⑥
      .
      .
      .
      END
```

(Continued on next page)

Calling System Services

Interpreting the Programming Examples

Figure 2-2 (Cont.) Interpreting MACRO Examples

FORTRAN Notes

- ❶ The module \$SYSSRVNAM in the FORTRAN system default library FORSYSDEF.TLB contains INTEGER and EXTERNAL declarations for each of the system services, so you need not provide any in your program explicitly. Other modules in FORSYSDEF declare the structures, offsets, and symbolic values used to refer to the data structures and symbolic return codes used by the system services.
- ❷ The EQUIV_NAME declaration allocates 255 bytes for the output data. The system service requires a descriptor for the equivalence name (output) argument. Since FORTRAN passes character values by descriptor as the default, there is no need to use the %DESCR actual argument function.
- ❸ The NAMLEN declaration allocates 2 bytes for the actual length of the returned equivalence name (an output argument). Since the \$TRNLNM service takes this argument by reference, and by reference is the default FORTRAN mechanism for numeric arguments, there is no need to use the %REF actual argument function.
- ❹ The status value returned by the system services must be treated as a 4 byte integer in FORTRAN programs. The declaration of the service names in the \$SYSSRVNAM module insures this for the service names themselves. However, variables used to hold status values should also be declared as INTEGER*4 quantities. Unpredictable results could occur if a REAL variable is used to hold return status values because of erroneous REAL to INTEGER conversions of the status value.
- ❺ Call the service as a function reference.

Enclose the arguments in parentheses. Specify a comma for each optional argument that you omit (including trailing arguments).

Since the SYS\$TRNLNM service requires its last argument to be passed by value, you must use the %VAL argument list function to force the compiler to use this mechanism.
- ❻ It is usually best to test the status return value immediately for success or failure. When used in a logical test, failure status values test as .FALSE. and success status values test as .TRUE. If it is important for your program to recognize precise return status values, you should use the LIB\$MATCH_COND library function to make sure that only the bits specific to the returned status are tested, and the "message info" bits are not part of the test. By using the value of the return status as the argument to the EXIT system supplied subroutine, you can cause your program to issue the exit status as its result value and test for this value in a DCL command procedure. Many other possibilities exist, including providing a condition handler to resolve the error condition without terminating the program. See the VAX FORTRAN User's Guide for more information about testing return status values.

(Continued on next page)

Calling System Services

Interpreting the Programming Examples

Figure 2-2 (Cont.) Interpreting MACRO Examples

PASCAL Equivalent

```

PROGRAM ORION ( OUTPUT );
TYPE
    $BYTE      = [BYTE] 1..255;
    $UWORD     = [WORD] 1..65535;
    BUFFER_TYPE = PACKED ARRAY [1..255] OF CHAR;
CONST
    LNM$STRING = 2      ①
[ASYNCHRONOUS] FUNCTION SYS$TRNLNM (      ①
    %REF ATTR :
        UNSIGNED := %IMMED 0;
    TABNAM :
        [CLASS_S] PACKED ARRAY
            [$12..$u2:INTEGER] OF CHAR;
    LOGNAM :
        [CLASS_S] PACKED ARRAY [$13..$u3:INTEGER]
            OF CHAR := %IMMED 0;
    %REF ACMODE :
        $BYTE := %IMMED 0;
    %REF ITMLST :
        [UNSAFE] ARRAY [$15..$u5:INTEGER] OF
            $BYTE := %IMMED 0) : INTEGER; EXTERN;
PROCEDURE ERROR;
BEGIN
    WRITELN (' Failed to translate logical name');
END;
VAR
    NAM_BUF   : ^BUFFER_TYPE; ③
    NAM_LEN   : ^INTEGER; ②
{
    NOTE: because of the various structures that an
    item can have, type compatibility is given up.
}
    ITEM_LIST : PACKED RECORD
        BUF_LEN   : $UWORD;
        ITEM_CODE : $UWORD;
        BUF_ADDR  : ^BUFFER_TYPE;
        LEN_ADDR  : ^INTEGER;
        TERMINATOR : INTEGER;
    END;
    IOCODE      : INTEGER;
    I           : INTEGER;
BEGIN
    ITEM_LIST.BUF_LEN      := 255;
    ITEM_LIST.ITEM_CODE    := LNM$STRING;
    ITEM_LIST.TERMINATOR   := 0;
    NEW( NAM_BUF );
    ITEM_LIST.BUF_ADDR     := NAM_BUF;
    NEW( NAM_LEN );
    ITEM_LIST.LEN_ADDR     := NAM_LEN;

```

(Continued on next page)

Calling System Services

Interpreting the Programming Examples

Figure 2-2 (Cont.) Interpreting MACRO Examples

```
IOCODE := SYS$TRNLNM( , 'LNM$SYSTEM', 'CYGNUS',  
                     , ITEM_LIST);  
IF NOT ODD( IOCODE )  
THEN  
    ERROR  
ELSE  
    BEGIN  
        WRITE( ' Logical name translates to ' );  
        FOR I := 1 TO NAM_LEN DO  
            WRITE( NAM_BUF[I] );  
        WRITELN( '' );  
    END;  
END.
```

PASCAL Notes

- ❶ The system service routine must be declared in an external function declaration in the function and procedure declaration section. Note that all the parameters for the system service call must be formally declared here.
- ❷ Specify the input character string directly to the system service call. The string descriptor is built by the compiler.
- ❸ The VAR declaration for the identifier NAM_BUF declares a pointer to a packed array of 255 characters for the associated name that is output from the system service. The packed array of characters is used as a string data type in PASCAL.
- ❹ The VAR declaration for the identifier NAM_LEN declares a pointer to a longword for the output length.
- ❺ Call the system service using the SYS\$ form of the service name. Enclose the arguments in parentheses and specify them in positional order. If you omit optional arguments, insert a comma as a placeholder. The default value for the argument will be supplied.
- ❻ The IF statement performs a logical test following the function reference to see if the service completed successfully. If an error or warning occurred during the service call, the procedure ERROR will be called.
- ❼ The value of the symbolic code LNM\$STRING is 2. Section 2.3 explains how to obtain values for symbolic codes.

3 Security Services

The VAX/VMS security system services provide various mechanisms that you can use to enhance the security of VAX/VMS systems. These services include facilities to:

- Create and maintain a rights database
- Create and translate access control list entries
- Modify a process rights list
- Check access protection
- Provide a security erase pattern for disks
- Control magnetic tape access

The following table identifies the system services related to system security.

Service Name	Function(s)
\$ADD_HOLDER	Adds holder record to rights database
\$ADD_IDENT	Adds identifier to rights database
\$ASCTOID	Translates identifier name to binary value
\$CHANGE_ACL	Creates or modifies an ACL
\$CHECK_ACCESS	Invokes system access protection check on behalf of another user
\$CHKPRO	Invokes system access protection check
\$CREATE_RDB	Initializes a rights database
\$ERAPAT	Generates a security erase pattern
\$FIND_HELD	Returns identifier(s) held by a holder in rights database
\$FIND_HOLDER	Returns holder(s) of an identifier in rights database
\$FINISH_RDB	Deallocates record stream and clears context value when searching the rights database
\$FORMAT_ACL	Formats ACE into a text string
\$GRANTID	Adds identifier to process or system rights list
\$IDTOASC	Translates identifier value to its identifier name
\$MOD_HOLDER	Modifies holder record in rights database
\$MOD_IDENT	Modifies identifier record in rights database
\$MTACCESS	Controls magnetic tape access
\$PARSE_ACL	Converts text ACE into binary format
\$REM_HOLDER	Deletes holder record from identifier's list of holders in rights database

Security Services

Service Name	Function(s)
\$REM_IDENT	Deletes identifier and all holders of that identifier from rights database
\$REVOKID	Removes identifier from process or system rights list

3.1 Overview of VAX/VMS Protection Scheme

The basis of the VAX/VMS security scheme is an *identifier*, which is a 32-bit binary value that represents a process to the system. An identifier can represent an individual user, a group of users, or some aspect of the environment in which a user is operating. A process is a *holder* of an identifier when that identifier can represent that process to the system.

The system *rights database* is an indexed file consisting of identifier and holder records. Those records define the identifiers and the holders of those identifiers on a system. When a process logs into the system, LOGINOUT creates a rights list for the process from the applicable entries in the rights database. Thus, a *process rights list* contains all the identifiers that the process holds. A process can be the holder of a number of identifiers. Each of those identifiers determines the identity and the access rights of the list holder. The process rights list becomes part of the process and is propagated to any created processes.

When a process attempts to access an object in the system, VAX/VMS uses the rights list when performing a protection check. The system compares the identifiers in the rights list to the protection attributes of the object and grants or denies access to the object based on the comparison. In other words, the entries in the rights list do not specifically grant access, instead the system uses them to perform a protection check when the process attempts to access an object.

The VAX/VMS protection scheme provides security with the mechanism of the access control list (ACL). An ACL consists of access control list entries (ACEs) that specify the type of access an identifier has to an object like a file, device, or mailbox. When a process attempts to access an object with an associated ACL, the system grants or denies access based on whether an exact match for the identifier in the ACL exists in the rights database.

The following sections describe each of the components of the security scheme—identifiers, rights database, process rights list, and ACLs—and the system services affecting those components.

3.2 Identifiers

The basic component of the VAX/VMS protection scheme is an identifier. This 32-bit binary value represents various types of agents using the system. The types of agents represented include individual users, groups of users, and environments in which a process is operating.

3.2.1 Identifier Format

Identifiers have two formats in the rights database: UIC format and ID format. The high-order bits of the identifier value specify the format of the identifier. Two high-order zero bits identify a UIC format identifier; bit 31, set to 1, identifies an ID format identifier.

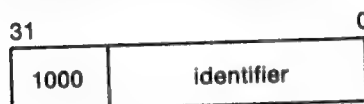
Each UIC identifier is unique and represents a system user. The UIC identifier contains the two high order bits that designate format, a member field, and a group field. Member numbers range from 0 to 65,534; group numbers range from 1 to 16,382.



UIC Format

ZK-1905-84

Bit 31, set to 1, specifies ID format. Bits 30 through 28 are reserved by DIGITAL. The remaining bits specify the identifier value.



ID Format

ZK-1906-84

3.2.2 Identifier Names

To the system, an identifier is a binary value; however, to make identifiers easy to use, the system translates the binary identifier value into an identifier name. The binary value and the identifier name are associated in the rights database.

An identifier name consists of 1 to 31 alphanumeric characters and contains at least one nonnumeric character. An identifier name cannot consist entirely of numeric characters. It can include the characters A through Z, dollar signs (\$) and underscores (_), as well as the numbers 0 through 9. Any lowercase characters are automatically converted to uppercase.

3.2.3 System-Defined Identifiers

System-defined identifiers, or environmental identifiers, are automatically defined when the rights database is initialized. The following system-defined identifiers correspond directly with the login classes and relate to the environment in which the process operates.

BATCH	All attempts at access made by batch jobs
NETWORK	All attempts at access made over the DECnet-VAX network
INTERACTIVE	All attempts at access made by interactive processes
LOCAL	All attempts at access made by users logged in at local terminals

Security Services

Identifiers

DIALUP	All attempts at access made by users logged in at dialup terminals
REMOTE	All attempts at access made by users logged in on a network

Depending on the environment in which the process is operating, LOGINOUT includes one or more of these identifiers when creating the process rights list.

3.2.4 General Identifiers

You can define general identifiers to meet the specific needs of your site. You grant these identifiers to users by establishing holder records in the rights database. General identifiers can identify a single user, a single UIC group, a group of users, or a number of groups.

You define identifiers and their holders in the rights database with the Authorize Utility or with the appropriate system services. You can define an identifier in the rights database to allow users from different UIC groups to hold an identifier. This allows you to create a different kind of group designation than the one used with the user's UIC.

The alternative grouping described here permits each user to be a member of multiple overlapping groups. In addition, you can control the types of access more closely than with UIC-based protection.

You can also define identifiers to represent particular terminals, times of days, or other site-specific environmental attributes. These identifiers are not given holder records in the rights database but may be granted to users by customer-written privileged software. This feature of the security system allows each site flexibility and (since the identifiers can be so specific to the site) enhanced security. See Section 3.3.2.4 for a programming example demonstrating this technique. Also, for more information, see the *Guide to VAX/VMS System Security*.

3.2.5 Identifier Attributes

An identifier has attributes associated with it in the rights database. Part of the process rights list includes the attributes of any identifiers that the process holds. A holder of an identifier can hold an attribute only if the identifier holds the attribute.

Attributes which may be added to identifiers include the DYNAMIC and RESOURCE attributes. The DYNAMIC attribute allows unprivileged holders of an identifier to add or remove the identifier from the process rights list. The RESOURCE attribute allows the holder of an identifier to charge resources, like disk blocks, to an identifier. Conversely, a holder who does not have the RESOURCE attribute cannot charge resources to the identifier, and an unprivileged holder who does not have the DYNAMIC attribute cannot modify the identifier.

The following example demonstrates the advantages of defining an identifier and holder(s) for a project:

The physics department of a school may have a common library with an associated disk quota on the system. (If disk quotas are in use, you must establish a quota file entry for the identifier to allow anyone to charge space to it.) You want to allow the faculty to charge any disk quota that they use in conjunction with the library to an identifier, and to prevent the students from charging resources to that identifier. You could define an identifier PHYSICS in the rights database with the holders FRED, a faculty member, and GEORGE, a student. If you can specify the RESOURCE attribute for FRED, that holder can charge resources to the PHYSICS identifier; if you do not specify the RESOURCE attribute for GEORGE, that holder cannot charge resources to the PHYSICS identifier.

3.3 Rights Database

The rights database is an indexed file containing two types of records that define all identifiers: identifier records and holder records.

One identifier record appears in the rights database for each identifier. The identifier record associates the identifier name with its 32-bit binary value, and specifies the attributes of the identifier. The following figure depicts the format of the identifier record.

IDENTIFIER VALUE
ATTRIBUTES
0
0
IDENTIFIER NAME

ZK-1904-84

One holder record exists in the rights database for each holder of each identifier. The holder record associates the holder with the identifier, specifies the attributes of the holder, and identifies the UIC identifier of the holder. The format of a holder record is:

IDENTIFIER VALUE
ATTRIBUTES
UIC IDENTIFIER OF HOLDER
(RESERVED)

ZK-1907-84

The rights database is an indexed file with three keys. The primary key is the identifier value, the secondary key is the holder id, and the third key is the identifier name. Through the use of the secondary key of the holder id, all the rights held by a process can be retrieved quickly when LOGINOUT creates the process rights list.

Security Services

Rights Database

3.3.1 Initializing a Rights Database

The rights database is initialized in one of the following ways:

- When a system is installed or upgraded
- With the Authorize Utility
- With the \$CREATE_RDB system service

When you call \$CREATE_RDB, you can use the **sysid** argument to pass the system identification value associated with the rights database. If you omit **sysid**, the system uses the current system time in 64-bit format. If the rights database already exists, \$CREATE_RDB fails with the error code RMS\$_FEX. To create a new rights database when one already exists, you must explicitly delete or rename the old one.

When a rights database is initialized, it is equated to the logical name RIGHTSLIST, which you must define as a system logical name at executive mode. If the logical name does not exist, the rights database is given the default file specification of SYS\$SYSTEM:RIGHTSLIST.DAT.

When created, RIGHTSLIST.DAT has the default protection of (S:RWED,O:RWED,G:RWE,W:R). World read access to the directory in which the database will be located is required so that all users can read the records in the database. In order to use \$CREATE_RDB, write access to the database is necessary. If the database is in SYS\$SYSTEM, which is the default, you need SYSPRV privilege to grant write access to the database.

When \$CREATE_RDB initializes a rights database, system-defined identifiers, which describe the environment in which a process can operate, are automatically created.

To add any other identifiers to the rights database you must define them with the Authorize Utility or with the appropriate system service.

3.3.2 Using System Services to Affect a Rights Database

The identifier and holder records in the rights database contain the following elements:

- Identifier binary value
- Identifier name
- Holder(s) of each identifier
- Attribute of each identifier and each holder of each identifier

You can use the Authorize Utility or one of the following system services to add, delete, display, modify, or translate the various elements of the rights database.

Action	Element	Service Used
Translate	Id name to id binary value	\$ASCTOID
	Id binary value to id name	\$IDTOASC
Add	New identifier record	\$ADD_HOLDER

Security Services

Rights Database

Action	Element	Service Used
Find	Identifier to holder record	\$ADD_IDENT
	Identifier value held by holder	\$FIND_HELD
	Holder(s) of an identifier	\$FIND_HOLDER
	All identifiers	\$IDTOASC
Modify	Attribute in holder record	\$MOD_HOLDER
	Attribute in identifier record	\$MOD_IDENT
Delete	Holder from identifier record	\$REM_HOLDER
	Identifier and all its holder(s)	\$REM_IDENT

When using these services, you need the following access. Note that if the rights database is in SYS\$SYSTEM, which is the default, you need SYSPRV privilege to grant write access to the database.

Service	Required Access
\$ADD_HOLDER	Write access
\$ADD_IDENT	Write access
\$ASCTOID	Read access
\$CREATE_RDB	Write access
\$FIND_HELD	Read access
\$FIND_HOLDER	Read access
\$FINISH_RDB	Read access
\$IDTOASC	Read access
\$MOD_HOLDER	Write access
\$MOD_IDENT	Write access
\$REM_HOLDER	Write access
\$REM_IDENT	Write access

3.3.2.1

Translating Identifier Names and Binary Values

An identifier, to the system, is a 32-bit binary value; however, to make identifiers easy to use each binary value has an associated identifier name. The id value and the ASCII id name string are associated in the rights database. You can use the \$ASCTOID and \$IDTOASC system services to translate from one format to another. When you pass the address of a string descriptor pointing to an identifier name to \$ASCTOID, the corresponding id binary value is returned. Conversely, you use the \$IDTOASC service to translate a binary id value to an ASCII id name string.

You can also use the \$IDTOASC service to list the identifier names of all of the identifiers in the rights database. Specify the **id** argument as -1, initialize the **context** argument to 0, and repeatedly call \$IDTOASC until the status code SS\$_NOSUCHID is returned. \$IDTOASC returns the identifier names in alphabetical order. When SS\$_NOSUCHID is returned, \$IDTOASC clears the context longword and deallocates the record stream. If you complete your calls to \$IDTOASC before SS\$_NOSUCHID is returned, use \$FINISH_RDB to clear the context longword and to deallocate the record stream.

Security Services

Rights Database

The following programming example uses \$IDTOASC to identify all identifiers in a rights database.

```
Program ID_LIST
*
* Produce a list of all the identifiers
*
integer SYS$IDTOASC
external SS$NORMAL, SS$NOSUCHID
character*31 NAME
integer IDENTIFIER, ATTRIBUTES
integer ID/-1/, LENGTH, CONTEXT/0/
integer NAME_DSC(2)/31, 0/
integer STATUS

*
* Initialization
*
NAME_DSC(2) = %loc(NAME)
STATUS = %loc(SS$NORMAL)

*
* Scan through the entire RDB ...
*
do while (STATUS .and. (STATUS .ne. %loc(SS$NOSUCHID)))
    STATUS = SYS$IDTOASC(%val(ID), LENGTH, NAME_DSC,
        IDENTIFIER, ATTRIBUTES, CONTEXT)
    if (STATUS .and. (STATUS .ne. %loc(SS$NOSUCHID))) then
        NAME(LENGTH+1:LENGTH+1) = ','
        print 1, NAME, IDENTIFIER, ATTRIBUTES
        1 format(1X,'Name: ',A31,' Id: ',Z8,', Attributes: ',Z8)
    end if
end do

*
* Do we need to finish the RDB ???
*
if (STATUS .ne. %loc(SS$NOSUCHID)) then
    call SYS$FINISH_RDB(CONTEXT)
end if
end
```

3.3.2.2 Adding Identifiers and Holders to Rights Database

You add identifiers to the rights database with the \$ADD_IDENT service in a program. When you call \$ADD_IDENT, use the **name** argument to pass the identifier name you want to add. You can specify an id value with the **id** argument; however, if you do not specify a value, the system selects an id value from the general identifier space.

In addition to defining the id value and id name, you use \$ADD_IDENT to specify attributes in the identifier record. Attributes are valid for a holder of an identifier *only* when they are set in both the identifier record and the holder record. The **attrib** argument is a longword containing a bit mask specifying the attributes. The symbol KGB\$V_RESOURCE, defined in the system macro library \$KGBDEF, sets the RESOURCE bit in the attribute longword, and the symbol KGB\$V_DYNAMIC sets the DYNAMIC bit. (You can use the prefix KGB\$M rather than KGB\$V.)

When \$ADD_IDENT successfully completes execution, a new identifier record exists in the rights database containing the identifier value, the identifier name, and the attributes of the identifier.

Once the identifier record exists in the rights database, you define the holder(s) of that identifier with the \$ADD_HOLDER system service. You pass the binary identifier value with the **id** argument; you specify the holder with the **holder** argument, which is the address of a quadword data structure with the following format.

UIC Identifier of Holder
0

ZK-1903-84

In the rights database, the holder identifier is in UIC format. You specify the attributes of the holder with the **attrib** argument in the same manner as with \$ADD_IDENT. Attributes are valid for a holder of an identifier only when they are set in both the identifier record and the holder record.

After \$ADD_HOLDER completes execution, a new holder record exists in the rights database containing the binary value of the identifier that the holder holds, the attributes of the holder, and the UIC of the holder.

3.3.2.3 Determining Holders of Identifiers

You can determine the holder(s) of a particular identifier with the \$FIND_HOLDER service in a program. When you call \$FIND_HOLDER, use the **id** argument to pass the binary value of the identifier whose holder you want to determine. On successfully completing execution, \$FIND_HOLDER returns the holder identifier with the **holder** argument and the attributes of the holder with the **attrib** argument.

You can identify all of the identifier's holders by initializing the **context** argument to 0, and repeatedly calling \$FIND_HOLDER as detailed in Section 3.3.3. Since \$FIND_HOLDER identifies the records by the same key (holder id), it returns the records in the order in which they were written.

3.3.2.4 Determining Identifiers Held

You can determine the identifier(s) held by a holder with the \$FIND_HELD service in a program. When you call \$FIND_HELD, use the **holder** argument to specify the holder whose identifier is to be found.

On completing execution, \$FIND_HELD returns identifier's binary id value and attributes.

You can identify all the identifiers held by the specified holder by initializing the **context** argument to zero, and repeatedly calling \$FIND_HELD as detailed in Section 3.3.3. Since \$FIND_HELD identifies the records by the same key (identifier), it returns the records in the order in which they were written.

The following programming example uses \$FIND_HELD to determine if a user is the holder of a particular identifier. This example also demonstrates how to define an identifier to represent particular terminals.

Security Services

Rights Database

```
.title SECURE_TERMINAL

; This module verifies that the user is executing this program
; from one of a set of "secure terminals" as outlined in file:
; SECURITY$:SECURE_TERMINAL.DATA;1.
;

.psect SECURE_TERMINAL, LONG

; The full names of all "secure terminals" are stored in
; file SECURITY$:SECURE_TERMINAL.DATA;1, which is an indexed file
; containing only the names of the secure terminals. If a name
; is found in that file, it is considered "secure".
;

.align LONG
XAB:  $XABKEY pos = 0, -
      siz = 64

.align LONG
FAB:  $FAB  fnm = <SECURITY$:SECURE_TERMINAL.DATA;1>, -
      fac = <GET>, -
      shr = <GET, PUT, UPD, DEL>, -
      org = IDX, -
      rfm = FIX, -
      mrs = 64, -
      xab = XAB

.align LONG
RAB:  $RAB  fab = FAB, -
      kbf = BUFFER, -
      ksz = 64, -
      ubf = BUFFER, -
      usz = 64, -
      rac = KEY

;
; For the identifiers we need these
;
NAME:
LENGTH: .blk1 1
        .address      BUFFER
BUFFER: .blkb 64
HOLDER: .blk1 1
        .long 0
ID_NAME: .ascid /SECURE_TERMINAL/
ID:      .blk1 1
CONXTY: .long 0
HELD:   .blk1 1

;
; In order to get the name of the particular terminal, we need this item list
;

.align LONG
ITMLST: .word 64
        .word JPI$_TERMINAL
        .address      BUFFER
        .address      LENGTH
        .long 0
```

Security Services

Rights Database

```

ABORT_: jmp      ABORT
:
: Here we go ...
:
: .entry SECURE_TERMINAL, ~n<>
:
: First, get the full device name
:
$GETJPIW_S      itmlst = ITMLST
blbc    r0, ABORT_
$OPEN    fab = FAB
blbc    r0, ABORT_
$CONNECT      rab = RAB
blbc    r0, ABORT_
$GET      rab = RAB
blbc    r0, ABORT_
:
: If we have gotten here, then our terminal is secure
:
$DISCONNECT      rab = RAB
$CLOSE    fab = FAB
:
: Is this user allowed to use the secure terminals
: (a holder of the SECURE_TERMINAL identifier)?
:
MOVW    $JPI$USERNAME, ITMLST+2
$GETJPIW_S      itmlst = ITMLST
pushal  LENGTH
pushal  NAME
pushal  NAME
calls   #3, STR$TRIM
$ASCTOID_S      name = NAME, id = HOLDER
$ASCTOID_S      name = ID_NAME, id = ID
$1:    $FIND_HELD_S      holder = HOLDER, id = HELD, ctxt = CONTEXT
blbc    r0, ABORT
cmpl    ID, HELD
bneq    $1
:
: Now pass control on to the program
:
calls   #0, @MAIN_PROGRAM_PROPER
$EXIT_S R0
.weak  MAIN_PROGRAM_PROPER
:
: Else kick the user out
:
: .external    LIB$SIGNAL
LIB$SIGNAL_:
: .address      LIB$SIGNAL
ABORT:  pushl    $SS$NOPRIV
        pushl    #0
        pushl    r0
        calls   #3, @LIB$SIGNAL_
        $EXIT_S $SS$NORMAL
        .end    SECURE_TERMINAL

```

Security Services

Rights Database

3.3.2.5 Modifying the Identifier Record

You modify an identifier record by changing the identifier's name, value, and/or attributes in the rights database with the `$MOD_IDENT` service in a program. Use the `id` argument to pass the binary value of the identifier whose record you want to modify. To enable attributes, use the `set_attr` argument, which is a longword containing a bit mask specifying the attributes. The symbol `KGB$V_RESOURCE`, defined in the system macro library `$KGBDEF`, sets the `RESOURCE` bit in the attribute longword, and the symbol `KGB$V_DYNAMIC` sets the `DYNAMIC` bit. (You can use the prefix `KGB$M` rather than `KGB$V`.)

If you want to disable the attributes for the identifier, use the `clr_attr`, which is a longword containing a bit mask specifying the attributes. If the same attribute is specified in `set_attr` and `clr_attr`, the attribute is enabled.

You can also change the identifier name and/or value with the `new_name` and `new_value` arguments. `New_name` is the address of a descriptor pointing to the identifier name string; `new_value` is a longword containing the binary id value. If you change the value of an identifier that is the holder of other identifiers (a UIC, for example) `$MOD_IDENT` updates all the corresponding holder records with the new holder id value.

When `$MOD_IDENT` successfully completes execution, a new identifier record exists in the rights database containing the identifier value, the identifier name, and the attributes of the identifier.

3.3.2.6 Modifying a Holder Record

You modify a holder record with the `$MOD_HOLDER` service in a program. When you call `$MOD_HOLDER`, use the `id` argument and the `holder` argument to pass the binary id value and the UIC holder identifier whose holder record you want to modify.

Use the `$MOD_HOLDER` service to enable or disable the attributes of an identifier in the same way as with `$MOD_IDENT`.

When `$MOD_HOLDER` completes execution, a new holder record exists in the rights database containing the identifier value, the identifier name, and the attributes of the identifier.

The following programming example uses `$MOD_HOLDER` to modify holder records in the rights database.

```
Program MOD_HOLDER
*
* Modify the attributes of all the holders of identifiers to reflect
* the current attribute setting of the identifiers themselves
*
external SS$_NOSUCHID
parameter KGB$M_RESOURCE = 1, KGB$M_DYNAMIC = 2
integer SYS$IDTOASC, SYS$FIND_HELD, SYS$MOD_HOLDER
*
```

Security Services

Rights Database

```

* Store information about the holder here
*
integer HOLDER(2)/2*0/
equivalence (HOLDER(1), HOLDER_ID)
integer HOLDER_NAME(2)/31, 0/
integer HOLDER_ID, HOLDER_CTX/0/
character*31 HOLDER_STRING

*
* Store attributes here
*
integer OLD_ATTR, NEW_ATTR, ID_ATTR, CONTEXT

* Store information about the identifier here
*
integer IDENTIFIER, ID_NAME(2)/31, 0/
character*31 ID_STRING
integer STATUS

*
* Initialize the descriptors
*
HOLDER_NAME(2) = %loc(HOLDER_STRING)
ID_NAME(2) = %loc(ID_STRING)

* Scan through all the identifiers
*
do while
+ (SYS$IDTOASC(%val(-1),, HOLDER_NAME, HOLDER_ID,, HOLDER_CTX)
+ .ne. %loc(SS$NOSUCHID))

* Test all the identifiers held by this identifier (our HOLDER)
*
if (HOLDER_ID .le. 0) go to 2
CONTEXT = 0
do while
+ (SYS$FIND_HELD(HOLDER, IDENTIFIER, OLD_ATTR, CONTEXT)
+ .ne. %loc(SS$NOSUCHID))

* Get name and attributes of held identifier
*
STATUS = SYS$IDTOASC(%val(IDENTIFIER),, ID_NAME,, ID_ATTR,)

* Modify the holder record to reflect the state of the identifier itself
*
if ((ID_ATTR .and. KGB$M_RESOURCE) .ne. 0) then
STATUS = SYS$MOD_HOLDER
+ (%val(IDENTIFIER), HOLDER, %val(KGB$M_RESOURCE),)
NEW_ATTR = OLD_ATTR .or. KGB$M_RESOURCE
else
STATUS = SYS$MOD_HOLDER
+ (%val(IDENTIFIER), HOLDER,, %val(KGB$M_RESOURCE))
NEW_ATTR = OLD_ATTR .and. (.not. KGB$M_RESOURCE)
end if
if ((ID_ATTR .and. KGB$M_DYNAMIC) .ne. 0) then
STATUS = SYS$MOD_HOLDER
+ (%val(IDENTIFIER), HOLDER, %val(KGB$M_DYNAMIC),)
NEW_ATTR = OLD_ATTR .or. KGB$M_DYNAMIC
else
STATUS = SYS$MOD_HOLDER
+ (%val(IDENTIFIER), HOLDER,, %val(KGB$M_DYNAMIC))
NEW_ATTR = OLD_ATTR .and. (.not. KGB$M_DYNAMIC)
end if

```

Security Services

Rights Database

```
*
* Were we successful ?
*
      if (.not. STATUS) then
        NEW_ATTR = OLD_ATTR
        call LIB$SIGNAL(%val(STATUS))
      end if
*
* Report it all
*
      print 1, HOLDER_STRING, ID_STRING,
        OLD_ATTR, ID_ATTR, NEW_ATTR
1      format(1X, 'Holder: ', A31, ' Id: ', A31,
        ' Old: ', Z8, ' Id: ', Z8, ' New: ', Z8)
      end do
2      continue
    end do
  end
```

3.3.2.7 Removing Identifiers and Holders from the Rights Database

To remove an identifier and *all of its holders*, use the \$REM_IDENT service in a program. When you call \$REM_IDENT, use the **id** argument to pass the binary value of the identifier that you want to remove. When \$REM_IDENT completes execution, the identifier and all of its associated holder records are removed from the rights database.

To remove a holder from the list of an identifier's holder, use the \$REM_HOLDER service in a program. When you call \$REM_HOLDER, use the **id** argument and the **holder** argument to pass the binary id value and the UIC identifier of the holder whose holder record you want to delete.

On successfully completing execution, \$REM_HOLDER removes the holder from the list of the identifier's holders.

3.3.3 Searching Operations

You can search the entire rights database when using the \$IDTOASC, \$FIND_HELD, and \$FIND_HOLDER services. You initialize the context longword to 0, and repeatedly call one of the three services until the status code SS\$_NOSUCHID is returned. When SS\$_NOSUCHID is returned, the service clears the context longword and deallocates the record stream. If you complete your calls to one of these services before SS\$_NOSUCHID is returned, you must use \$FINISH_RDB to clear the context longword and to deallocate the record stream.

The structure of the rights database affects the order in which each of these services returns the records when you search the rights database. The rights database is an indexed file with three keys. The primary key is the identifier binary value, the secondary key is the holder UIC identifier, and the third key is the identifier name.

During a searching operation, the service obtains the first record with an indexed RMS GET operation. The key used for the GET operation depends on the service. \$FIND_HOLDER uses the identifier binary value; \$FIND_HELD uses the holder UIC identifier. After the indexed GET, the service returns the records with sequential RMS GET operations. Consequently, the file organization, the key used for the first GET operation, and the order in which the records were originally written in the database determine how the service returns records in a searching operation.

Security Services

Rights Database

Service	Record Order
\$IDTOASC	Identifier name order.
\$FIND_HELD	First GET operation—holder key. Subsequent records are returned in the order in which they were written.
\$FIND_HOLDER	First GET operation—identifier key. Subsequent records are returned in the order in which they were written.

The following programming example uses \$IDTOASC, \$FINISH_RDB, and \$FIND_HOLDER to search the entire rights database for identifiers with holders and produces a list of those identifiers and their holders.

```
Module ID_HOLDER
( main = MAIN,
  addressing_mode(external=GENERAL) ) =
begin
!
!   Produce a list of all the identifiers, which have holders,
!   with their respective holders.
!
!
!   Declarations:
!
library
'SYS$LIBRARY:LIB';
forward routine
MAIN;
external routine
LIB$PUT_OUTPUT,
SYS$FAO,
SYS$IDTOASC,
SYS$FINISH_RDB,
SYS$FIND_HOLDER;
!
!   To create static descriptors
!
macro S_DESCRIPTOR[NAME, SIZE] =
own
  %name(NAME, '_BUFFER'): block[%number(SIZE), byte],
  %name(NAME): block[DSC$K_S_BLN, byte]
  preset( [DSC$B_CLASS] = DSC$K_CLASS_S,
    [DSC$W_LENGTH] = %number(SIZE),
    [DSC$A_POINTER] = %name(NAME, '_BUFFER') ); %;
!
!   Descriptors for ID, holder NAME, and output LINE
!
S_DESCRIPTOR('ID_NAME', 31);
S_DESCRIPTOR('NAME', 31);
S_DESCRIPTOR('LINE', 76);
own
  STATUS,
  ID,
  ID_LENGTH,
  ID_CONTEXT: initial(0),
  HOLDER,
  LENGTH,
  CONTEXT: initial(0),
  ATTRIBS,
  VALUE,
  LINE_: block[DSC$K_S_BLN, byte]
  preset( [DSC$B_CLASS] = DSC$K_CLASS_S,
    [DSC$A_POINTER] = LINE_BUFFER );
```


Security Services

Rights Database

```
!
!       To check for existence of an ID or HOLDER
!
macro CHECK(EXPRESSION) =
    (STATUS = %remove(EXPRESSION)) and (.STATUS neq SS$_NOSUCHID) %;
!
!       List all the identifiers, which have holders, with their holders
!
routine MAIN =
begin
!
!       Examine all IDs (-1)
!
while
    CHECK(<SYS$IDTOASC(-1, ID_LENGTH, ID_NAME, ID, ATTRIBS, ID_CONTEXT)>)
do
    begin
        CONTEXT = 0;
!
!       Find all holders of ID
!
        while CHECK(<SYS$FIND_HOLDER(.ID, HOLDER, ATTRIBS, CONTEXT)>) do
            begin
!
!       Translate the HOLDER to find its NAME
!
                SYS$IDTOASC(.HOLDER, LENGTH, NAME, VALUE, ATTRIBS, 0);
!
!       Print a message reporting ID and HOLDER
!
                SYS$FAO( %ascid'Id: !AD, Holder: !AD',
                    LINE_[DSC$W_LENGTH], LINE,
                    .ID_LENGTH, .ID_NAME[DSC$A_POINTER],
                    .LENGTH, .NAME[DSC$A_POINTER] );
                LIB$PUT_OUTPUT(LINE_);
            end;
        end;
    return SS$_NORMAL;
end;
end
eludem
```

3.4 Creating, Translating, and Maintaining ACEs

An access control list (ACL) is a list of entries defining the type of access allowed to an object in the system like a file, device, or mailbox. When a process attempts to access an object with an associated ACL, the system allows access based on the type of access specified by the entries in the ACL.

Access control list entries (ACEs), to the system, are in binary form; however, to make ACEs easy to use, they have text string format. You use \$FORMAT_ACL and \$PARSE_ACL to translate ACEs from one format to another in the same way that \$IDTOASC and \$ASCTOID translate identifiers from binary to text format and text to binary format.

You create and manipulate ACLs with the ACL editor, the DCL command SET ACL, and the \$CHANGE_ACL system service in a program.

Security Services

Creating, Translating, and Maintaining ACEs

3.4.1 Format of ACEs Types

There are four types of ACEs:

- Alarm
- Application dependent
- Default protection
- Identifier

The alarm ACE provides a security alarm when an object is accessed in a particular way. The application ACE contains application dependent or user-defined information. The default protection ACE defines the default protection for a directory so that that protection can be propagated to the files and subdirectories created in that directory. The identifier ACE controls the type of access allowed to a particular user or group of users as specified by an identifier.

An ACE's type determines its format. The following sections describe the format of each of the four types of ACEs. Symbols specifying byte offsets and type values are defined in the system macro library (\$ACEDEF).

3.4.1.1

Alarm ACE

The access alarm ace sets a security alarm on an object in the system. The following figure illustrates its format.

flags	type	length
access		
alarm name		

ZK-1710-84

Field	Symbol Name	Description
Length	ACE\$B_SIZE	Byte containing the length in bytes of the ACE buffer
Type	ACE\$B_TYPE	Byte containing the type value ACE\$C_ALARM
Flags	ACE\$W_FLAGS	Word containing alarm ACE information and ACE type-independent information
Access	ACE\$L_ACCESS	Longword containing a mask indicating the access modes to be watched
Alarm Name	ACE\$T_AUDITNAME	Counted character string containing the alarm name

The flag word contains information specific to alarm ACEs and information applicable to all types of ACEs. In the flags word, the first byte contains flags specific to each ACE type; the second byte contains flags common to all ACE types. The following symbols are bit offsets to the alarm ACE information.

Security Services

Creating, Translating, and Maintaining ACEs

Bit	Meaning When Set
ACE\$V_SUCCESS	Indicates that the alarm is raised when access is successful
ACE\$V_FAILURE	Indicates that the alarm is raised when access fails

The following symbols are bit offsets to ACE information that is applicable to all types of ACEs.

Table 3-1 ACE Type-Independent Information

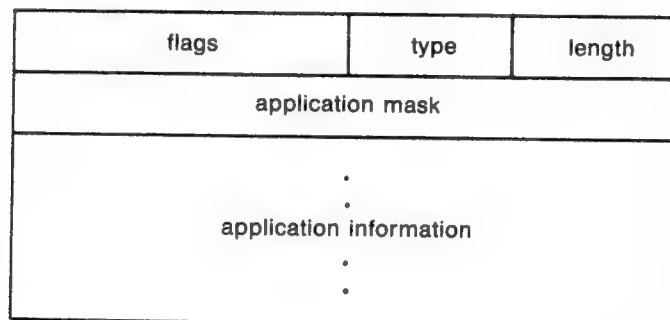
Bit	Meaning When Set
ACE\$V_DEFAULT	This ACE is added to the ACL of any file created in the directory whose ACL contains this ACE. This option is applicable only for an ACE in a directory file's ACL.
ACE\$V_HIDDEN	This ACE is application dependent. The DCL ACL commands and the ACL editor cannot be used to change the setting; the DCL command DIRECTORY /ACL does not display it.
ACE\$V_NOPROPAGATE	This ACE is not propagated among versions of the same file.
ACE\$V_PROTECTED	This ACE is not deleted if the entire ACL is deleted; instead this ACE must be explicitly deleted.

The following symbol values are offsets to bits within the access mask. You can also obtain the symbol values as masks with the appropriate bit set using the prefix ACE\$M rather than ACE\$V.

Bit	Meaning When Set
ACE\$V_READ	Read access is monitored.
ACE\$V_WRITE	Write access is monitored.
ACE\$V_EXECUTE	Execute access is monitored.
ACE\$V_DELETE	Delete access is monitored.
ACE\$V_CONTROL	Modification of the access field is monitored.

3.4.1.2 Application ACE

The application ACE contains application dependent information. The following figure illustrates its format.



ZK-1711-84

Security Services

Creating, Translating, and Maintaining ACEs

Field	Symbol Name	Description
Length	ACE\$B_SIZE	Byte containing the length in bytes of the ACE buffer.
Type	ACE\$B_TYPE	Byte containing the type value ACE\$C_INFO.
Flags	ACE\$W_FLAGS	Word containing application ACE information and ACE type-independent information.
Application Mask	ACE\$L_INFO_FLAGS	Longword containing a mask defined and used by the application.
Application Information	ACE\$T_INFO_START	Variable length data structure defined and used by the application. The length of this data is implied by length field.

The flag word contains information specific to application ACEs and information applicable to all types of ACEs. In the flags word, the first byte contains flags specific to each ACE type; the second byte contains flags common to all ACE types. For details on the ACE type-independent information see Table 3-1. The following symbols are bit offsets to the application ACE information.

Bit	Meaning When Set
ACE\$V_INFO_TYPE	Four-bit field containing a value indicating whether the application is a CSS application (ACE\$C_CSS), a customer application (ACE\$C_CUST), or a VMS application (ACE\$C_VMS)

3.4.1.3

Default Protection ACE

The default protection ACE specifies the default protection for all files and subdirectories created in the directory. This type of ACE can be used only in the ACL of a directory file. The following figure illustrates its format.

flags	type	length
spare		
system		
owner		
group		
world		

ZK-1712-84

Security Services

Creating, Translating, and Maintaining ACEs

Field	Symbol Name	Description
Length	ACE\$B_SIZE	Byte containing the length in bytes of the ACE buffer.
Type	ACE\$B_TYPE	Byte containing the type value ACE\$C_DIRDEF.
Flags	ACE\$W_FLAGS	Word containing ACE type-independent information.
Spare	ACE\$L_SPARE1	Longword that is reserved for future use and must be zero.
System	ACE\$L_SYS_PROT	Longword containing a mask indicating the access mode granted to system users. Each bit represents one type of access.
Owner	ACE\$L_OWN_PROT	Longword containing a mask indicating the access mode granted to the owner. Each bit represents one type of access.
Group	ACE\$L_GRP_PROT	Longword containing a mask indicating the access mode granted to group users. Each bit represents one type of access.
World	ACE\$L_WOR_PROT	Longword containing a mask indicating the access mode granted to the world. Each bit represents one type of access.

The flag word contains the ACE type-independent information. In the flags word, the first byte contains flags specific to each ACE type; the second byte contains flags common to all ACE types. For details, see Table 3-1.

The system interprets the bits within the access mask as shown in the following list. The symbol values are offsets to bits within the mask indicating the access mode granted in the system, owner, group, and world fields.

Bit	Meaning When Set
ACE\$V_READ	Read access is granted.
ACE\$V_WRITE	Write access is granted.
ACE\$V_EXECUTE	Execute access is granted.
ACE\$V_DELETE	Delete access is granted.

You can also obtain the symbol values as masks with the appropriate bit set by using the prefix ACE\$M rather than ACE\$V.

3.4.1.4 Identifier ACE

The identifier ACE controls the type of access allowed based on identifiers. Access is controlled on whether or not an exact match exists in the process rights list for the identifier(s) in the ACE. The following figure illustrates its format.

Security Services

Creating, Translating, and Maintaining ACEs

flags	type	length
access		
reserved		
reserved		
.		
.		
.		
Identifier		
Identifier		
.		
.		
.		

ZK-1713-84

Field	Symbol Name	Description
Length	ACE\$B_SIZE	Byte containing the length in bytes of the ACE buffer.
Type	ACE\$B_TYPE	Byte containing the type value ACE\$C_KEYID.
Flags	ACE\$W_FLAGS	Word containing identifier ACE information and ACE type-independent information.
Access	ACE\$L_ACCESS	Longword containing a mask indicating the access mode granted to the specified identifiers.
Reserved	ACE\$V_RESERVED	Longwords containing application-specific information. The number of reserved longwords is specified in the flags field.
Identifier	ACE\$L_KEY	Longwords containing identifiers. The number of longwords is implied by ACE\$B_LENGTH. If an accessor holds all the listed identifiers, the ACE is said to match the accessor and the access specified in ACE\$L_ACCESS is granted.

The flag word contains information specific to identifier ACEs and information applicable to all types of ACEs. In the flags word, the first byte contains flags specific to each ACE type; the second byte contains flags common to all ACE types. For details on the ACE type-independent information, see Table 3-1. The following symbol is a bit offset to identifier ACE information.

Security Services

Creating, Translating, and Maintaining ACEs

Bit	Meaning When Set
ACE\$V_RESERVED	Four-bit field containing the number of longwords to reserve for application dependent data. The number must be between 0 and 15. The reserved longwords, if any, immediately precede the identifiers.

The following symbol values are offsets to bits within the mask indicating the access mode granted in the system, owner, group, and world fields.

Bit	Meaning When Set
ACE\$V_READ	Read access is granted.
ACE\$V_WRITE	Write access is granted.
ACE\$V_EXECUTE	Execute access is granted.
ACE\$V_DELETE	Delete access is granted.
ACE\$V_CONTROL	Modification of the access field is granted.

You can also obtain the symbol values as masks with the appropriate bit set by using the prefix ACE\$M rather than ACE\$V.

3.4.2 Translating ACEs

You use the \$FORMAT_ACL service to translate ACEs from binary format into a text string. The **aclent** argument is the address of a descriptor pointing to a buffer containing the description of the ACE. The first byte of the buffer contains the length of the ACE; the second byte contains the type, which in turn defines the format of the ACE. The following four values specify ACE type.

Value	ACE Type
ACE\$C_ALARM	Alarm ACE
ACE\$C_INFO	Application ACE
ACE\$C_DIRDEF	Default Protection ACE
ACE\$C_KEYID	Identifier ACE

The **acllen** argument specifies the length of the text string written to the buffer pointed to by **aclstr**. You use the **width**, **trmdsc**, and **indent** arguments to specify a particular width, termination character, and number of blank characters for an ACE. **Accnam** contains the address of an array of 32 quadword descriptors that define the names of the bits in the access mask of the ACE. If **accnam** is omitted, the following names are used:

Bit 0	READ
Bit 1	WRITE
Bit 2	EXECUTE
Bit 3	DELETE
Bit 4	CONTROL
Bit 5	BIT_5
Bit 6	BIT_6
.	.
.	.
Bit 31	BIT_31

Security Services

Creating, Translating, and Maintaining ACEs

The `$PARSE_ACL` service translates an ACE from text string format to binary format. The `aclstr` argument is the address of a string descriptor pointing to the ACE text string. As with `$FORMAT_ACL`, the `aclent` is the address of a descriptor pointing to a buffer containing the description of the ACE. The first byte of the buffer contains the length of the ACE; the second byte contains the type, which in turn defines the format of the ACE. If `$PARSE_ACL` fails, the `errpos` argument points to the failing point in the string. `Accnam` contains the address of an array of 32 quadword descriptors that define the names of the bits in the access mask of the ACE. If `accnam` is omitted, the names specified in the description of `$FORMAT_ACL` are used.

3.4.3 Creating and Maintaining ACEs

You use the `$CHANGE_ACL` service to create or modify an ACL associated with a system object. You specify the object whose ACL is to be modified with either the `chan` argument, which specifies the I/O channel associated with the object, or with the `objnam` argument, which specifies the object name. If you specify `objnam`, `chan` must be omitted or specified as zero. `Objtyp` specifies the type of object. The values specifying object type are:

<code>ACL\$C_DEVICE</code>	Object is a device
<code>ACL\$C_FILE</code>	Object is a Files-11 structure level 2 file
<code>ACL\$C_GROUP_GLOBAL_SECTION</code>	Object is a group global section
<code>ACL\$C_LOGICAL_NAME_TABLE</code>	Object is a logical name table
<code>ACL\$C_SYSTEM_GLOBAL_SECTION</code>	Object is a system global section

Use the `acmode` argument to specify the access mode used when checking file access protection. By default, kernel mode is used, but the system compares `acmode` against the caller's access mode and uses the least privileged mode. `Itmlst` is an item list specifying the changes to be made to the ACL. Each item code consists of three elements. The following figure illustrates the format of the item code.

code	buflen
bufadr	
unused	

ZK-1701-84

The item list ends with a longword containing the value zero. `Buflen` contains the number of bytes in the buffer containing information passed to or from `$CHANGE_ACL` pointed to by `bufadr`. The third longword of the standard item descriptor is not used by `$CHANGE_ACL` and should be zero.

The item code specifies the change to be made to the ACL. The following symbols for the item codes are defined in the system macro library (`$ACLDEF`).

Security Services

Creating, Translating, and Maintaining ACEs

ACL\$_ACLENGTH	Returns the size, in bytes, of the object's ACL. Bufadr points to a longword that contains the size.
ACL\$_ADDACLENT	Adds an ACE to the beginning of the ACL when contxt is 0, to the end of the ACL when contxt is -1, or at a location pointed to by a prior ACL\$_FNDACETYP or ACL\$_FNDACLENT. Bufadr points to a variable length data structure containing the ACE to be added. You can add more than one ACE with ACL\$_ADDACLENT; however, buflen must contain the total size of all ACEs contained in the buffer.
ACL\$_DELACLENT	Deletes the ACE pointed to by bufadr or if bufadr is specified as zero, the ACE pointed to by a prior ACL\$_FNDACETYP or ACL\$_FNDACLENT.
ACL\$_DELETEACL	Deletes the entire ACL with the exception of protected ACEs.
ACL\$_FNDACETYP	Locates an ACE of the type pointed to by bufadr .
ACL\$_FNDACLENT	Locates the ACE pointed to by bufadr .
ACL\$_RLOCK_ACL	Obtains a read lock on an object in order to lock out all writers to the object's ACL. Regardless of the caller's mode, ACL locks are user mode locks so that all access modes will interlock ACLs correctly.
ACL\$_WLOCK_ACL	Obtains an exclusive lock on an object in order to lock out all other readers and writers to the object's ACL. Regardless of the caller's mode, ACL locks are user mode locks so that all access modes will interlock ACLs correctly.
ACL\$_MODACLENT	Replaces the ACE pointed to by a prior ACL\$_FNDACETYP or ACL\$_FNDACLENT with the ACE pointed to by bufadr .
ACL\$_READACE	Reads the ACE pointed to by ACL\$_FNDACETYP or ACL\$_FNDACLENT into the buffer pointed to by bufadr .
ACL\$_READACL	Reads the object's ACL. Contxt should be initially set to zero. Complete ACEs are read into the buffer pointed to by bufadr .
ACL\$_UNLOCK_ACL	Releases the lock obtained with ACL\$_RLOCK_ACL or ACL\$_WLOCK_ACL.

When you add an ACE with ACL\$_ADDACLENT or locate an ACE with ACL\$_FNDACETYP or ACL\$_FNDACLENT, \$CHANGE_ACL searches the ACL for a match for the ACE in the ACE buffer. \$CHANGE_ACL does not always make a match based on the entire ACE buffer; instead, the ACE type determines how \$CHANGE_ACL makes a match.

- A default protection ACE (ACE\$_DIRDEF) matches only on the type field (ACE\$_TYPE). An ACL can have only one default protection ACE because \$CHANGE_ACL stops searching once it locates a match.
- An identifier ACE (ACE\$_KEYID) matches on the flags (ACE\$_FLAGS) and identifier (ACE\$_KEY) fields.
- An alarm ACE (ACE\$_ALARM) matches on the flags (ACE\$_FLAGS) and access mask (ACE\$_ACCESS) fields.

Security Services

Creating, Translating, and Maintaining ACEs

- All other ACE types match on the entire ACE buffer.

Because \$CHANGE_ACL uses these matching rules, adding an ACE sometimes results in the replacement of another ACE. For example, if you add an identifier ACE with the same flags and identifier fields but a different access mask, the new ACE replaces the old ACE. When you add an ACE on the top of an ACL, \$CHANGE_ACL deletes any matching ACE since it will not be seen. If you add an ACE below a matching ACE in an ACL, the added ACE has no effect since it will not be seen.

The following programming example uses \$CHANGE_ACL to add an ACE to the ACL of a terminal. (See Section 3.6 for a related example.)

```
Module SECURE (main = MAIN, addressing_mode(external-general)) =
begin
!
!   Insert a record into the specified terminal's ACL so that
!   holders of the SECURE_TERMINAL identifier may do confidential
!   work with that terminal.
!
!   To use: $ SECURE tt20:
!
!   Confidential applications will, of course, need to use
!   SYS$CHKPRO to verify that users are authorized to use them.
!
library
  'SYS$LIBRARY:LIB';
forward routine
  MAIN;
external routine
  LIB$GET_FOREIGN,      ! To get the name of the terminal
  SYS$CHANGE_ACL,      ! To make the actual changes to the ACL
  SYS$PARSE_ACL;       ! To translate the ACE from ASCII
compiletime
  POSITION = 0;
macro
!
!   Some of the routines require dynamic string descriptors
!
DYNAMIC_DESCRIPTOR =
  block[DSC$K_D_BLN, byte]
  preset( [DSC$B_CLASS] = DSC$K_CLASS_D, [DSC$B_DTYPE] = 0,
    [DSC$W_LENGTH] = 0, [DSC$A_POINTER] = 0 ) %,
!
!   These two macros are used solely for initializing the access name table
!
INITIALIZE(BIT_NUMBER, BIT_NAME) =
  [BIT_NUMBER, DSC$W_LENGTH] = %charcount(BIT_NAME),
  [BIT_NUMBER, DSC$A_POINTER] = uplit byte(BIT_NAME) %,
IGNORE(START, FINISH)[] =
  %if START leq FINISH %then
    [START, DSC$W_LENGTH] =
      %charcount(%string('BIT_', START)),
    [START, DSC$A_POINTER] =
      uplit byte(%string('BIT_', START))
  %if START les FINISH %then , %fi
  %assign(POSITION, START+1)
  IGNORE(%number(POSITION), FINISH)
  %fi %;
```

Security Services

Creating, Translating, and Maintaining ACEs

```
own
    STATUS,
    OBJNAM: DYNAMIC_DESCRIPTOR,          ! The name of this terminal
    BUFADR: block[ACL$S_ADDACLENT, byte], ! The new ACE
    ACLENT: block[DSC$K_D_BLN, byte]
        preset( [DSC$W_LENGTH] = ACL$S_ADDACLENT,
                [DSC$A_POINTER] = BUFADR ),
    ITMLST: $ITMLST_DECL(),
!
!     The Access Name Table:
!
!     Here we specify the ASCII names of all the access types
!
    ACCNAM: blockvector[32, DSC$K_S_BLN, byte]
        preset( INITIALIZE( 0, 'READ',
                            1, 'WRITE',
                            2, 'LOGICAL',
                            3, 'PHYSICAL',
                            4, 'CONTROL',
                            5, 'CONFIDENTIAL' ), ! Our hero !
                IGNORE(6, 31) );
!
!     Prompt the user for the terminal's name
!     Create a new ACE
!     Add the ACE to the ACL of the terminal
!
routine MAIN =
begin
    LIB$GET_FOREIGN(OBJNAM, %ascii'Device: ');
    SYS$PARSE_ACL(%ascii'(IDENTIFIER=SECURE_TERMINAL,ACCESS=CONFIDENTIAL)',
                  ACLENT, 0, ACCNAM);
    $ITMLST_INIT( itmlst = ITMLST,
                  ( itmcod = ACL$C_ADDACLENT,
                    bufsiz = .BUFADR[ACE$B_SIZE],
                    bufadr = BUFADR ) );
    if not
        (STATUS = SYS$CHANGE_ACL(0, %ref(ACL$C_DEVICE), OBJNAM, ITMLST, 0,0,0))
    then
        signal_stop(.STATUS);
    return SS$NORMAL;
end;
end
eludom
```

3.5 Modifying a Rights List

When a process is created, LOGINOUT builds a rights list for the process consisting of the identifiers that the user holds and any appropriate environmental identifiers. A system rights list is a default rights list used in addition to any process rights list. Modifications to the system rights list effectively become modifications to the rights of each process.

A privileged subsystem can alter the process or system rights list with the \$GRANTID or \$REVOKID services. These services are not intended for the general system user. \$GRANTID adds an identifier to a rights list, or if the identifier is already part of the rights list, it modifies the attributes of the identifier. \$REVOKID removes an identifier from a rights list. If the identifier, specified by either **id** or **name**, is the holder of any other identifiers, the identifier is removed from those holder records.

Security Services

Modifying a Rights List

`$GRANTID` and `$REVOKID` treat the `pidadr` and `prcnam` arguments in the same way as all other process control services.

See the *Guide to VAX/VMS System Security* for more details.

You may also modify the process or system rights list with the DCL command `SET RIGHTS_LIST`. Additionally, you can use `SET RIGHTS_LIST` to modify the attributes of the identifier if the identifier is already part of the rights list. Note that you may not use the `SET RIGHTS_LIST` command to modify the rights database from which the rights list was created. See the *VAX/VMS DCL Dictionary* for more information about using the `SET RIGHTS_LIST` command.

3.6 Checking Access Protection

VAX/VMS provides two system services which check access to objects on the system: `SY$CHKPRO` and `SY$CHECK_ACCESS`. `SY$CHKPRO` performs the system access protection check on a user attempting direct access to an object on the system; `SY$CHECK_ACCESS` performs a similar check but on behalf of a third-party accessor attempting access to an object. These services are described below.

3.6.1 SY\$CHKPRO

The `$CHKPRO` service invokes the access protection check that is used by the system. The service does not grant or deny access, rather it performs the protection check on behalf of a layered product, application program, or other similar subsystem which in turn must specifically grant or deny access.

You pass the input and output information to `$CHKPRO` with the `itmlst` argument, which is the address of an item list of descriptors. `$CHKPRO` compares the item list of the rights and privileges of the accessor to a list of the protection attributes of the object to be accessed. If the accessor can access the object, `$CHKPRO` returns the status `SS$_NORMAL`; if the accessor cannot access the object, `$CHKPRO` returns the status `SS$_NOPRIV`. `$CHKPRO` does not grant or deny access. The subsystem itself must grant or deny access based on the output (`SS$_NORMAL` or `SS$_NOPRIV`) from `$CHKPRO`.

`$CHKPRO` also returns an item list of the rights or privileges that allowed the accessor access to the object as well as the names of security alarms raised by the access attempt. For information on the item codes defined for `$CHKPRO`, see the description of `$CHKPRO` in Part II of this manual.

The following programming example uses `$CHKPRO` to verify that a user is authorized to use a terminal for confidential work. `$CHKPRO` does not explicitly grant access; it simply performs the protection check. The application itself must grant or deny access based on the output from `$CHKPRO`. See Section 3.4.3 for a related example.

Security Services

Checking Access Protection

```

Module CHECK (main = MAIN, addressing_mode(external=general)) =
begin
  library
    'SYS$LIBRARY:LIB';
  forward routine
    MAIN;
  external routine
    SYS$CHKPRO,
    SYS$CHANGE_ACL,
    LIB$GET_VM;

  DWN
    STATUS,
    ACLENGTH,
    ACL: ref block[, byte],
    ITMLST1: $ITMLST_DECL(),
    ITMLST2: $ITMLST_DECL(items=2);

  routine MAIN =
  begin
    !
    !       Query for the size of the user terminal's ACL
    !
    $ITMLST_INIT( itmlst = ITMLST1,
      ( itmcd = ACL$C_ACLENGTH, bufadr = ACLENGTH ) );
    SYS$CHANGE_ACL(0, %ref(ACL$C_DEVICE), %ascid'TT:', ITMLST1, 0,0,0);
    !
    !       Allocate memory to store the ACL
    !
    LIB$GET_VM(%ref(.ACLENGTH), ACL);
    !
    !       Read the entire ACL into the buffer
    !
    $ITMLST_INIT( itmlst = ITMLST1,
      ( itmcd = ACL$C_READACL, bufadr = .ACL, bufsiz = .ACLENGTH ) );
    SYS$CHANGE_ACL(0, %ref(ACL$C_DEVICE), %ascid'TT:', ITMLST1, 0,0,0);
    !
    !       Check the object for CONFIDENTIAL (BIT_5) access
    !
    $ITMLST_INIT( itmlst = ITMLST2,
      ( itmcd = CHP$_ACL, bufadr = .ACL, bufsiz = .ACLENGTH ),
      ( itmcd = CHP$_ACCESS, bufadr = uplit(%b'100000') ) );
    if not (STATUS = SYS$CHKPRO(ITMLST2)) then
      signal_stop(.STATUS);
    return SS$NORMAL;
  end;
end
eludom

```

3.6.2 SYS\$CHECK_ACCESS

Whereas SYS\$CHKPRO performs the system access protection check on a user attempting access to an object, SYS\$CHECK_ACCESS executes the protection check on behalf of a third-party accessor. An example of this would be a file server program that uses SYS\$CHECK_ACCESS to ensure that an accessor (the third party) requesting a file has the required privileges to do so.

Security Services

Checking Access Protection

You pass the input and output information to `$CHECK_ACCESS` with the `itmlst` argument, which is the address of an item list of descriptors. You also pass the name of the accessor and the name and type of the object being accessed with the arguments `usrnam`, `objnam` and `objtyp`, respectively. `$CHECK_ACCESS` compares the rights and privileges of the accessor to a list of the protection attributes of the object to be accessed. If the accessor can access the object, `$CHECK_ACCESS` returns the status `SS$_NORMAL`; if the accessor cannot access the object, `$CHECK_ACCESS` returns the status `SS$_NOPRIV`.

`$CHECK_ACCESS` does not grant or deny access. The subsystem itself must grant or deny access based on the output (`SS$_NORMAL` or `SS$_NOPRIV`) from `$CHECK_ACCESS`.

`$CHECK_ACCESS` also returns an item list of the rights or privileges that allowed the accessor access to the object as well as the names of security alarms raised by the access attempt. For information on the item codes defined for `$CHECK_ACCESS`, see the description of `$CHECK_ACCESS` in Part II of this manual.

3.7 Additional Security Services

VAX/VMS provides two additional services that affect system security. The `$ERAPAT` service provides a consistent mechanism by which users can write a security erase pattern for disks. The security erase patterns can be custom configured to fit the individual needs of a site. The `$MTACCESS` service checks the accessibility field in a magnetic tape label to determine if a volume is VMS protected.

For more information, see the descriptions of the `$MTACCESS` and the `$ERAPAT` services in Part II of this manual.

4

Event Flag Services

Event flags are status posting bits maintained by VAX/VMS for general programming use. Programs can use event flags to perform a variety of signaling functions. Event flag services clear, set and read event flags. They also can place a process in a wait state pending the setting of an event flag or flags. The following system services are event flag services.

- Associate Common Event Flag Cluster (\$ASCEFC)
- Disassociate Common Event Flag Cluster (\$DACEFC)
- Delete Common Event Flag Cluster (\$DLCEFC)
- Set Event Flag (\$SETEF)
- Clear Event Flag (\$CLREF)
- Read Event Flags (\$READEF)
- Wait for Single Event Flag (\$WAITFR)
- Wait for Logical OR of Event Flags (\$WFLOR)
- Wait for Logical AND of Event Flags (\$WFLAND)

Some system services set an event flag to indicate the completion or the occurrence of an event; the calling program can test the flag. The following are some of the system services that use event flags to signal events to the calling process.

- Enqueue Lock Request (\$ENQ and \$ENQW)
- Get Device/Volume Information (\$GETDVI and \$GETDVIW)
- Get Job/Process Information (\$GETJPI and \$GETJPIW)
- Get System-wide Information (\$GETSYI and \$GETSYIW)
- Queue I/O Request (\$QIO and \$QIOW)
- Set Timer (\$SETIMR)
- Update Section File on Disk (\$UPDSEC)
- Update Section File on Disk and Wait (\$UPDSECW)

Moreover, event flags can be used by more than one process as long as the cooperating processes are in the same group. Thus, if you have developed an application that requires the concurrent execution of several processes, you can use event flags to establish communication among them and to synchronize their activity.

Event Flag Services

Event Flag Numbers and Event Flag Clusters

4.1 Event Flag Numbers and Event Flag Clusters

Each event flag has a unique decimal number; event flag arguments in system service calls refer to these numbers. For example, if you specify event flag 1 in a call to the \$QIO system service, then event flag number 1 is set when the I/O operation completes.

To allow manipulation of groups of event flags, the flags are ordered in clusters, with 32 flags in each cluster, numbered from right to left, corresponding to bits 0 through 31 in a longword. The clusters are also numbered from 0 to 3. The range of event flag numbers encompasses the flags in all clusters: event flag 0 is the first flag in cluster 0, event flag 32 is the first flag in cluster 1, and so on.

There are two types of clusters, local event flag clusters and common event flag clusters.

- A local event flag cluster can only be used internally by a single process. Local clusters are automatically available to each process.
- A common event flag cluster can be shared by cooperating processes in the same group. Before a process can refer to a common event flag cluster, it must explicitly "associate" with the cluster. Association is described in Section 4, Common Event Flag Clusters.

The ranges of event flag numbers and the clusters to which they belong are summarized in Table 4-1.

Table 4-1 Summary of Event Flag and Cluster Numbers

Cluster Number	Event Flag Numbers	Description	Restriction
0	0-31	Process-local event flag clusters for general use	Event flags 24 through 31 are reserved for system use
1	32-63		
2	64-95	Assignable common event flag cluster	Must be associated before use
3	96-127		

4.1.1 Specifying Event Flag and Event Flag Cluster Numbers

The same system services manipulate flags in both local and common event flag clusters. Since the event flag number implies the cluster number, you do not have to specify the cluster number when you call a system service that refers to an event flag.

When a system service requires an event flag cluster number as an argument, you need only specify the number of any event flag that is in the cluster. Thus, to read the event flags in cluster 1, you could specify any number in the range 32 through 63.

Event Flag Services

Event Flag Numbers and Event Flag Clusters

To prevent accidental use of an event flag that is already in use elsewhere in your program, you should allocate and deallocate local event flags. The *VAX/VMS Run-Time Library Routines Reference Manual* describes routines you can use to allocate an arbitrary event flag (`LIB$GET_EF`), to allocate a particular event flag (`LIB$RESERVE_EF`) or, to deallocate an event flag (`LIB$FREE_EF`), from the process-wide pool of available local event flags. No similar routines exist for common event flags.

4.2 Examples of Event Flag Services

Local event flags are most commonly used in conjunction with other system services. For example, you can use the Set Timer (`$SETIMR`) system service to request that an event flag be set at a specific time of day or after a specific interval of time has passed. If you want to place a process in a wait state for a specified period of time, you could specify an event flag number for the `$SETIMR` service and then use the Wait for Single Event Flag (`$WAITFR`) system service, as follows:

```
TIME: .BLKQ 1 ; will contain time interval to wait
.
.
$SETIMR_S - ; set the timer
    EFN=#33, -
    DAYTIM=TIME
$WAITFR_S -
    EFN=#33 ; wait until timer expires
```

In this example, the `daytim` argument refers to a 64-bit time value. Details on how to obtain a time value in the proper format for input to this service are contained in Section 8, Timer and Time Conversion Services.

4.2.1 Event Flag Waits

Three system services place the process in a wait state until an event flag, or group of event flags, is set.

- The Wait for Single Event Flag (`$WAITFR`) system service places the process in a wait state until a single flag has been set.
- The Wait for Logical OR of Event Flags (`$WFLOR`) system service places the process in a wait state until **any one** of a specified group of event flags has been set.
- The Wait for Logical AND of Event Flags (`$WFLAND`) system service places the process in a wait state until **all** of a specified group of event flags have been set.

Another system service that accepts an event flag number as an argument is the Queue I/O Request (`$QIO`) system service. The following example shows a program segment that issues two `$QIO` system service calls, and uses the `$WFLAND` system service to wait until both I/O operations complete before it continues execution.

Event Flag Services

Examples of Event Flag Services

```
$QIO_S EFN=#1,... ① ; issue first I/O request
BSBW ERROR ; check for error
$QIO_S EFN=#2,... ; issue second I/O request
BSBW ERROR ; check for error
$WFLAND_S - ② ; wait until both complete
EFN=#1, - ③
MASK=#B0110
BSBW ERROR ; check for error
; continue execution
```

- ① The event flag argument is specified in each \$QIO request. Both of these event flags are in cluster 0.
- ② After both I/O requests are successfully queued, the program calls the Wait for Logical AND of Event Flags (\$WFLAND) system service to wait until the I/O operations are completed. In this service call, the **efn** argument can specify any event flag number in the cluster containing the event flags to be waited for. The **mask** argument specifies that flags 1 and 2 are to be waited for.
- ③ Note that the \$WFLAND system service (and the other wait system services) wait for the event flag to be set; they do not wait for the I/O operation to complete. If some other event were to set the required event flags, the wait for event flag would complete prematurely. Use of event flags must be carefully coordinated. (See Section 7.3.1 for a discussion of the recommended technique for testing I/O completion.)

4.3 Setting and Clearing Event Flags

System services that use event flags clear the event flag specified in the system service call before they queue the timer or I/O request. This ensures that the state of the event flag is known by the process. If you are using event flags in local clusters for other purposes, be sure the flag's initial value is what you want before you use it.

The Set Event Flag (\$SETEF) and Clear Event Flag (\$CLREF) system services set and clear specific event flags. For example, the following system service call clears event flag 32.

```
$CLREF_S EFN=#32
```

The \$SETEF and \$CLREF services return successful status codes that indicate whether the specified flag was set or clear when the service was called. The caller can thus determine the previous state of the flag, if necessary. The codes returned are SS\$_WASSET and SS\$_WASCLR.

Event flags in a common event flag cluster are all initially clear when the cluster is created. The next section describes the creation of common event flag clusters.

4.4 Common Event Flag Clusters

Common event flags act as a communication link between images executing in different processes in the same group. Common event flags are often used as a synchronization tool for other more complicated communication techniques such as logical names and global sections. For more information on using event flags to synchronize communication between processes see Section 2.5.

Before any processes can use event flags in a common event flag cluster, the cluster must be created. The Associate Common Event Flag Cluster (\$ASCEFC) system service creates a common event flag cluster. Once a cluster has been created, other processes in the same group can call \$ASCEFC to establish their association with the cluster, so they can access flags in it.

When a common event flag cluster is created, it must be identified by a name string. (Section 4.7.1 explains the format of this string.) Each process that associates with the cluster must use the same name to refer to the cluster; the \$ASCEFC system service establishes correspondence between the cluster name and the cluster number that a process assigns to it.

The following example shows how a process might create a common event flag cluster named COMMON_CLUSTER and assign it a cluster number of 2.

```
CLUSTER:
    .ASCID /COMMON_CLUSTER/      ; cluster name
    .
    $ASCEFC_S -                  ; create cluster 2
        EFN=#05, -
        NAME=CLUSTER
```

Subsequently, other processes in the same group may associate with this cluster. Those processes must use the same character string name to refer to the cluster; however, the cluster numbers they assign do not have to be the same.

Common event flag clusters are either temporary or permanent. The **perm** argument to the \$ASCEFC system service defines whether the cluster is temporary or permanent.

Temporary clusters require an element of the creating process's quota for timer queue entries (TQELM quota). They are deleted when all processes associated with the cluster have disassociated. Disassociation can be performed explicitly, with the Disassociate Common Event Flag Cluster (\$DACEFC) system service, or implicitly, when the image exits.

Permanent clusters require the creating process to have the PRMCEB user privilege. They continue to exist until they are explicitly marked for deletion with the Delete Common Event Flag Cluster (\$DLCEFC) system service.

If every cooperating process that is going to use a common event flag cluster has the necessary privilege or quota to create a cluster, the first process to call the \$ASCEFC system service creates the cluster.

Event Flag Services

Disassociating and Deleting Common Event Flag Clusters

4.5 Disassociating and Deleting Common Event Flag Clusters

When a process no longer needs access to a common event flag cluster, it issues the Disassociate Common Event Flag Cluster (\$DACEFC) system service. When all processes associated with a temporary cluster have issued a \$DACEFC system service, the system deletes the cluster. If a process does not explicitly disassociate itself from a cluster, the system performs an implicit disassociation when the image that called \$ASCEFC exits.

Permanent clusters, however, must be explicitly marked for deletion with the Delete Common Event Flag Cluster (\$DLCEFC) system service. After the cluster has been marked for deletion, it is not deleted until all processes associated with it have been disassociated.

4.6 Example of Using a Common Event Flag Cluster

The following is an example of four cooperating processes that share a common event flag cluster. The processes named ORION, CYGNUS, LYRA, and PEGASUS are in the same group.

```
Process ORION
CNAME: .ASCID /TITUS/ ; descriptor for cluster name

① $ASCEFC__S -      ; create common cluster
    EFN=#64, -
    NAME=CNAME ②
    BSBW ERROR ; check for error

③ $WFLAND__S -
    EFN=#64, -
    MASK=#_B1110 ; wait for flags 1,2,3
    BSBW ERROR ; check for error
⑦ $DACEFC__S -
    EFN=#64 ; disassociate cluster

Process CYGNUS
ORION__FLAGS: .ASCID /TITUS/ ; descriptor for
                ; cluster name

① $ASCEFC__S -
    EFN=#64, -
    NAME=ORION__FLAGS
    BSBW ERROR ; check for error
    $SETEF__S - ; set event flag 1
    EFN=#65
    BSBW ERROR ; check for error
    $DACEFC__S - ; disassociate
    EFN=#64

Process LYRA
SHARE: .ASCID /TITUS/ ; descriptor for cluster name
```

Event Flag Services

Example of Using a Common Event Flag Cluster

```
❶ $ASCEFC__S - ; associate with cluster 3
    EFN=#96, -
    NAME=SHARE
BSBW ERROR ; check for error
$SETEF__S - ; set flag 3
    EFN=#99
BSBW ERROR ; check for error
$DACEFC__S - ; disassociate
    EFN=#96
```

Process PEGASUS

CLUSTER: .ASCID /TITUS/ ; descriptor for cluster name

```
❷ $ASCEFC__S - ; associate with cluster
    EFN=#64, -
    NAME=CLUSTER
BSBW ERROR ; check for error
$WAITFR__S - ; wait for flag 1
    EFN=#65
BSBW ERROR ; check for error
    ; continue

$SETEF__S - ; set flag 2
    EFN=#66
BSBW ERROR ; check for error
$DACEFC__S - ; disassociate
    EFN=#64
```

- ❶ Assume for this example that ORION is the first process to issue the \$ASCEFC system service and therefore is the creator of the cluster. Since this is a newly created cluster, all event flags in it are clear.
- ❷ The argument **name** in the \$ASCEFC system service call is a pointer to the descriptor CNAME for the name to be assigned to the cluster; in this example, the cluster is named TITUS. This service call associates this name with cluster 2 of process ORION, containing event flags 64 through 95. Cooperating processes CYGNUS, LYRA, and PEGASUS must use the same character string name to refer to this cluster.
- ❸ The continuation of process ORION depends on work done by processes CYGNUS, LYRA, and PEGASUS. The Wait For Logical AND of Event Flags (\$WFLAND) system service call specifies a mask indicating the event flags that must be set before process ORION can continue. The mask in this example (^B1110) indicates that the second, third, and fourth flags in the cluster must be set.
- ❹ Process CYGNUS executes, associates with the cluster, sets event flag 65 (flag 1 in the cluster), and disassociates.
- ❺ Process LYRA associates with the cluster, but instead of referring to it as cluster 2, it refers to it as cluster 3 (with event flags in the range 96 through 127). Thus, when process LYRA sets flag 99, it is setting flag number 3 in TITUS.
- ❻ Process PEGASUS associates with the cluster, waits for an event flag set by process CYGNUS, and sets an event flag itself.
- ❼ When all three event flags are set, process ORION continues execution and calls the \$DACEFC system service. Since ORION did not specify the **perm** argument when it created the cluster, TITUS is deleted.

Event Flag Services

Common Event Flag Clusters in Shared Memory

4.7 Common Event Flag Clusters in Shared Memory

A common event flag cluster in memory shared by multiple processors is a vehicle by which processes executing on different CPUs can communicate with each other. A process can create a common event flag cluster using the Associate Common Event Flag Cluster (\$ASCEFC) service, specifying a cluster name that locates the cluster in memory shared by multiple processors (see Section 4.7.1). Other processes on the same or a different processor can associate with that cluster by specifying the same cluster name.

The SHMEM user privilege is required to create or delete a common event flag cluster in memory shared by multiple processors, but it is not required to associate with an existing cluster.

4.7.1 Cluster Name

The **name** argument to the Associate Common Event Flag Cluster (\$ASCEFC) service identifies the cluster that the process is creating or associating with. The **name** argument specifies a descriptor pointing to a character string that determines whether the cluster is in memory shared by multiple processors. The format of this string is as follows:

`[shared-memory-name:]cluster-name`

- The **shared-memory-name** identifies the memory shared by multiple processors in which the cluster exists or is to be created. (This name was assigned when the memory unit was connected at system generation time.) If this field is not included, the cluster exists or is created in memory that is local to the processor on which the calling process is executing.
- The **cluster-name** is the name of the cluster. You may choose any valid name, from 1 to 15 characters; however, all processes associating with the same common event flag cluster must specify the same name.

If you wish, you can include both the **shared-memory-name** and the **cluster-name** for an event flag cluster in memory shared by multiple processors. However, if you want to use existing programs without recompiling or relinking, you can specify just a **cluster-name** and have the system translate it to a complete specification. The system attempts to perform logical name translation of the string specified by the **NAME** argument in the following manner.

- The current name string is searched for a colon. If a colon is found within the current name string, the event flag cluster is assumed to be located in shared memory and translation proceeds in the following manner.
 - 1 The portion of the current name string to the right of the colon is placed in the cluster-name buffer. The portion of the current name string to the left of the colon becomes the new current name string.
 - 2 CEF\$ is prefixed to the current name string and the result is subjected to logical name translation.
 - 3 If the result contains a logical name, steps 1 and 2 are repeated until translation does not succeed or until the number of translations performed exceeds the number specified by the SYSGEN parameter LNM\$C_MAXDEPTH.

Event Flag Services

Common Event Flag Clusters in Shared Memory

- 4 The CEF\$ prefix is stripped from the current name string that could not be translated. This name becomes the shared-memory name. The cluster name is the current string contained in the cluster-name buffer.
- If the event flag cluster is located in local memory, translation proceeds in the following manner.
 - 1 CEF\$ is prefixed to the current name string and the result is subjected to logical name translation.
 - 2 If the result is a logical name, step 1 is repeated until translation does not succeed or until the number of translations performed exceeds the number specified by the SYSGEN parameter LNM\$_MAXDEPTH.
 - 3 The CEF\$ prefix is stripped from the current name string that could not be translated. This current string is the cluster-name.

For example, assume that you have made the following logical name assignment:

```
$ DEFINE CEF$CLUS_RT SHRMEM$1:CLUS_RT
```

Assume also that your program contains the following statements:

```
NAMEDESC:
    .ASCID /CLUS_RT/      ; descriptor for logical name of cluster
    .
    $ASCEFC_S -
    ...,NAME=NAMEDESC,...
```

The following logical name translation takes place.

- 1 CEF\$ is prefixed to CLUS_RT.
- 2 CEF\$CLUS_RT is translated to SHRMEM\$1:CLUS_RT. (No further translation is successful. When logical name translation fails, the string is passed to the service.)

There are two exceptions to the logical name translation method discussed in this section.

- If the name string starts with an underscore (_), VAX/VMS strips the underscore and considers the resultant string to be the actual name (that is, no further translation is performed).
- If the name string is the result of a logical name translation, then the name string is checked to see if it has the "terminal" attribute. If the name string is marked with the "terminal" attribute, VAX/VMS considers the resultant string to be the actual name (that is, no further translation is performed).

4.8 Example of Using Event Flag Services

REPORT.FOR

```
! Associate common event flag cluster
STATUS = SYS$ASCEFC (XVAL(64),
2 'JESSIE',,,)
IF (.NOT. STATUS) CALL LIB$SIGNAL (XVAL(STATUS))
! Create subprocess to execute concurrently
MASK = IBSET (MASK,0)
```


Event Flag Services

Example of Using Event Flag Services

```
STATUS = LIB$SPAWN ('RUN REPORTSUB', ! image
2                'INPUT.DAT',      ! SYS$INPUT
2                'OUTPUT.DAT',     ! SYS$OUTPUT
2                MASK
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
! wait for response from subprocess
STATUS = SYS$WAITFR (%VAL(64))
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))

.
.
REPORTSUB.FOR
.
.
! Do operations necessary for
! continuation of parent process
.
.
! Associate common event flag cluster
STATUS = SYS$ASCEFC (%VAL(64),
2                'JESSIER',..)
IF (.NOT. STATUS)
2 CALL LIB$SIGNAL (%VAL(STATUS))
! Set flag for parent process to resume
STATUS = SYS$SETEF (%VAL(64))
.
.
```

Common event flags are often used for communicating between a parent process and a created subprocess. In this example, REPORT.FOR creates a subprocess to execute REPORTSUB.FOR, which performs a number of operations.

Once REPORTSUB.FOR performs its first operation, the two processes can perform in parallel. REPORT.FOR and REPORTSUB.FOR use the common event flag cluster named JESSIER to communicate.

REPORT.FOR associates the cluster name with a common event flag cluster, creates a subprocess to execute REPORTSUB.FOR, then waits for REPORTSUB.FOR to set the first event flag in the cluster. REPORTSUB.FOR performs its first operation, associates the cluster name JESSIER with a common event flag cluster, and sets the first flag. From then on, the processes execute concurrently.

5

AST (Asynchronous System Trap) Services

Some system services allow a process to request that it be interrupted when a particular event occurs. Since the interrupt occurs asynchronously (out of sequence) with respect to the process's execution, the interrupt mechanism is called an asynchronous system trap (AST). The trap provides a transfer of control to a user-specified procedure that handles the event.

The following system services are AST services.

- Set AST Enable (\$SETAST)
- Declare AST (\$DCLAST)
- Set Power Recovery AST (\$SETPRA)

The system services that use the AST mechanism accept as an argument the address of an AST service routine, that is, a routine to be given control when the event occurs.

Listed below are some of the services that use ASTs.

- Declare AST (\$DCLAST)
- Enqueue Lock Request (\$ENQ)
- Get Device/Volume Information (\$GETDVI)
- Get Job/Process Information (\$GETJPI)
- Get System-wide Information (\$GETSYI)
- Queue I/O Request (\$QIO)
- Set Timer (\$SETIMR)
- Set Power Recovery AST (\$SETPRA)
- Update Section File on Disk (\$UPDSEC)

For example, if you call the Set Timer (\$SETIMR) system service, you can specify the address of a routine to be executed when a time interval expires or at a particular time of day. The service schedules the execution of the routine and returns; the program image continues executing. When the requested timer event occurs, the system "delivers" an AST by interrupting the process and calling the specified routine.

The following example shows a typical program that calls the \$SETIMR system service with a request for an AST when a timer event occurs.

AST (Asynchronous System Trap) Services

```

NOON:      .BLKQ      1      ; will contain 12:00 system time
           .ENTRY     LIBRA,0 ; entry mask for LIBRA
           .
           .
1  $SETIMR_S -      ; set timer
           DAYTIM=NOON, -
           ASTADR=TIMEAST
BSBW      ERROR      ; check for error
           .
           .          +-----+
           .          | Timer   |
           .          | Interrupt! 2 |
           .          +-----+
           .ENTRY     TIMEAST,"M<>" ; entry mask for AST routine
           .          ; handle timer request
3  RET
           .END      LIBRA      ; done

```

- 1 The call to the `$SETIMR` system service requests an AST at 12:00 noon. The `DAYTIM` argument refers to the quadword `NOON`, which must contain the time in system time (64-bit) format. For details on how this is done, see Section 8, *Timer and Time Conversion Services*. The `ASTADR` argument refers to `TIMEAST`, the address of the AST service routine. When the call to the system service completes, the process continues execution.
- 2 The timer expires at 12:00 and notifies the system. The system interrupts execution of the process and gives control to the AST service routine.
- 3 The user routine `TIMEAST` handles the interrupt. When the AST routine completes, it issues a `RET` instruction to return control to the program. The program resumes execution at the point at which it was interrupted.

The following sections describe in more detail how ASTs work and how to use them.

5.1 Access Modes for AST Execution

Each request for an AST is associated with the access mode from which the AST is requested. Thus, if an image executing in user mode requests notification of an event by means of an AST, the AST service routine executes in user mode.

Since the ASTs you use will almost always execute in user mode, you do not need to be concerned with access modes. However, you should be aware of some system considerations for AST delivery. These considerations are described in Section 5.5, *AST Delivery*.

5.2 ASTS and Process Wait States

A process that is in a wait state can be interrupted for the delivery of an AST and the execution of an AST service routine. When the AST service routine completes execution, the process is returned to the wait state, if the condition that caused the wait is still in effect.

Any wait states can be interrupted, except suspended waits (SUSP) and suspended outswapped waits (SUSPO).

AST (Asynchronous System Trap) Services

ASTS and Process Wait States

5.2.1 Event Flag Waits

If a process is waiting for an event flag and is interrupted by an AST, the wait state is restored following execution of the AST service routine. If the flag is set at completion of the AST service routine (for example, by completion of an I/O operation) then the process continues execution when the AST service routine completes.

Event flags are described in detail in Section 5.3, Event Flag Services.

5.2.2 Hibernation

A process can place itself in a wait state with the Hibernate (\$HIBER) system service. This wait state can be interrupted for the delivery of an AST. When the AST service routine completes execution, the process continues hibernation. The process can, however, "wake" itself in the AST service routine or be awakened by another process or as the result of a timer-scheduled wake-up request. Then, it continues execution when the AST service routine completes.

Process suspension is another form of wait; however, a suspended process cannot be interrupted by an AST. Process hibernation and suspension are described in Section 8, Process Control Services.

5.2.3 Resource Waits and Page Faults

When a process is executing an image, the system can place the process in a wait state until a required resource becomes available, or until a page in its virtual address space is paged into memory. These waits, which are generally transparent to the process, can also be interrupted for the delivery of an AST.

5.3 How ASTS are Declared

Most ASTs occur as the result of the completion of an asynchronous event initiated by a system service (for example, a \$QIO or \$SETIMR request) when the process requests notification by means of an AST.

There is also a system service that creates ASTs, the Declare AST (\$DCLAST) system service. With this service, a process can declare an AST only for the same or for a less privileged access mode.

You may find occasional use for the \$DCLAST system service in your programming applications; you may also find the \$DCLAST service useful when you want to test an AST service routine.

AST (Asynchronous System Trap) Services

The AST Service Routine

5.4 The AST Service Routine

An AST service routine must be a separate procedure. The system calls the AST with a CALLG instruction; the routine must return using a RET instruction. If the service routine modifies any registers other than R0 or R1, it must set the appropriate bits in the entry mask so that the contents of those registers are saved.

Since it is impossible to know when the AST service routine will begin executing, you must take care when you write the AST service routine that it does not modify any data or instructions used by the main procedure (unless, of course, that is its function).

On entry to the AST service routine, the Argument Pointer register (AP) points to an argument list that has the following format:

31	8	7	0
0		5	
AST parameter			
R0			
R1			
PC			
PSL			

ZK-855-82

The registers R0 and R1, the PC, and the PSL in this list are those that were saved when the process was interrupted by delivery of the AST.

The AST parameter is an argument passed to the AST service routine so that it can identify the event that caused the AST. When you call a system service requesting an AST, or when you call the \$DCLAST system service, you can supply a value for the AST parameter. If you do not specify a value, it defaults to 0.

The following example illustrates an AST service routine. In this example, the ASTs are queued by the \$DCLAST system service; the ASTs are delivered to the process immediately, so that the service routine is called following each \$DCLAST system service call.

AST (Asynchronous System Trap) Services

The AST Service Routine

```

        .ENTRY CELESTEF,0                ; entry mask
1  $DCLAST_8 -                          ; AST with parameter=1
        ASTADR=ASTRTN, -
        ASTPRM=#1
        .
        $DCLAST_8 -                      ; AST with parameter=2
        ASTADR=ASTRTN, -
        ASTPRM=#2
        .
        RET                             ; return control
;
ASTRTN: .WORD 0                          ; entry mask
2  CMPL #1,4(AP)                        ; check if AST parameter=1
        BEQL 10$                       ; if equal, goto 10$
        CMPL #2,4(AP)                  ; check if AST parameter=2
        BEQL 20$                       ; if equal, goto 20$
10$:
        RET                             ; handle first AST
        ; return
20$:
        RET                             ; handle second AST
        ; return
        .
        .END CELESTEF

```

- 1 The program CELESTEF calls the \$DCLAST AST system service twice to queue ASTs. Both ASTs specify the AST service routine, ASTRTN. However, a different parameter is passed for each call.
- 2 The first action that this AST routine takes is to check the AST parameter, so that it can determine if the AST being delivered is the first or second one declared. The value of the AST parameter determines the flow of execution. If there are a number of different values determining a number of different paths of execution, it is recommended that you use the VAX MACRO instruction, CASE.

5.5 AST Delivery

When a condition causes an AST to be delivered, the system may not be able to deliver the AST to the process immediately. An AST *cannot* be delivered if any of the following conditions exists.

- An AST service routine is currently executing at the same or at a more privileged access mode.

Because ASTs are implicitly disabled when an AST service routine executes, one AST routine cannot be interrupted by another AST routine declared for the same access mode. It can, however, be interrupted for an AST declared for a more privileged access mode.

- AST delivery is explicitly disabled for the access mode.

A process can disable the delivery of AST interrupts with the Set AST Enable (\$SETAST) system service. This service may be useful when a program is executing a sequence of instructions that should not be interrupted for the execution of an AST routine.

- The process is executing at an access mode more privileged than that for which the AST is declared.

AST (Asynchronous System Trap) Services

AST Delivery

For example, if a user mode AST is declared as the result of a system service but the program is currently executing at a higher access mode (because of another system service call, for example), the AST is not delivered until the program is once again executing in user mode.

If an AST cannot be delivered when the interrupt occurs, the AST is queued until the condition(s) disabling delivery are removed. Queued ASTs are ordered by the access mode from which they were declared, with those declared from more privileged access modes at the front of the queue. If more than one AST is queued for an access mode, the ASTs are delivered in the order in which they are queued.

5.6 Example of Using AST Services

```
PROGRAM DISK_DOWN

implicit none
! Status variable
INTEGER STATUS

STRUCTURE /ITMLST/
  UNION
    MAP
      INTEGER*2 BUFLN,
2      CODE
      INTEGER*4 BUFADR,
2      RETLENADR
    END MAP
    MAP
      INTEGER*4 END_LIST
    END MAP
  END UNION
END STRUCTURE
RECORD /ITMLST/ DVILIST(2),
2      JPILIST(2)
! Information for GETDVI call
INTEGER PID_BUF,
2      PID_LEN
! Information for GETJPI call
CHARACTER*7 TERM_NAME
INTEGER TERM_LEN

EXTERNAL DVI$_PID,
2      JPI$_TERMINAL

! AST routine and flag
INTEGER AST_FLAG
PARAMETER (AST_FLAG = 2)
EXTERNAL NOTIFY_USER

INTEGER SYS$GETDVIW,
2      SYS$GETJPI,
2      SYS$WAITFR

! set up for SYS$GETDVI
DVILIST(1).BUFLN = 4
DVILIST(1).CODE = %LOC(DVI$_PID)
DVILIST(1).BUFADR = %LOC(PID_BUF)
DVILIST(1).RETLENADR = %LOC(PID_LEN)
DVILIST(2).END_LIST = 0

! find PID number of process using SYS$DRIVED
STATUS = SYS$GETDVIW (,
2
2      '_MTAO:',      ! device
2      DVILIST,      ! item list
2      ...)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
```

AST (Asynchronous System Trap) Services

Example of Using AST Services

```
! Get terminal name and fire AST
JPILIST(1).CODE = %LOC(JPI$_TERMINAL)
JPILIST(1).BUFLen = 7
JPILIST(1).BUFADR = %LOC(TERM_NAME)
JPILIST(1).RETLENADR = %LOC(TERM_LEN)
JPILIST(2).END_LIST = 0
STATUS = SYS$GETJPI (,
2          PID_BUF,          !process id
2          ,
2          JPILIST,          !itemlist
2          ,
2          NOTIFY_USER,      !AST
2          TERM_NAME)        !AST arg
IF (.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))
! ensure that AST was executed
STATUS = SYS$WAITFR(%VAL(AST_FLAG))
IF (.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))

END

SUBROUTINE NOTIFY_USER (TERM_STR)
! AST routine that broadcasts a message to TERMINAL

! dummy argument
CHARACTER*(*) TERM_STR

CHARACTER*8 TERMINAL
INTEGER LENGTH

! status variable
INTEGER STATUS

CHARACTER*(*) MESSAGE
PARAMETER (MESSAGE =
2          'SYS$TAPE going down in 10 minutes')

! flag to indicate AST executed
INTEGER AST_FLAG

! declare system routines
INTRINSIC LEN
INTEGER SYS$BRDCST,
2      SYS$SETEF
EXTERNAL SYS$BRDCST,
2      SYS$SETEF,
2      LIB$SIGNAL

! Add underscore to device name
LENGTH = LEN (TERM_STR)
TERMINAL(2:LENGTH+1) = TERM_STR
TERMINAL(1:1) = '_'

! Send message
STATUS = SYS$BRDCST(MESSAGE,
2          TERMINAL(1:LENGTH+1))
IF (.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))

! set event flag
STATUS = SYS$SETEF (%VAL(AST_FLAG))
IF (.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))

END
```

The VAX FORTRAN program above is an example of a program that finds the pid number of any user that is using a particular disk and fires an AST to notify the user that the disk is coming down.

6 Logical Name Services

The VAX/VMS logical name services provide a technique for manipulating and substituting character string names. Logical names are commonly used to specify devices or files for input or output operations. You can use logical names to communicate information between processes by creating a logical name in one process in a shared logical name table, and translating the logical name in another process. The VAX/VMS logical name services are as follows:

- Create Logical Name (\$CRELNM)
- Create Logical Name Table (\$CRELNT)
- Delete Logical Name (\$DELLNM)
- Translate Logical Name (\$TRNLNM)

This section describes how to use system services to establish logical names for general application purposes. The system performs special logical name translation procedures for names associated with I/O services and with services that can deal with facilities located in shared (multiport) memory; for further information see the following sections:

- Device names for I/O services: Section 7 in this manual and the discussion of logical names in the *VAX/VMS DCL Dictionary*
- Common event flag cluster names: Section 4
- Mailbox names: Section 7
- Global section names: Section 11

6.1 Logical Name Concepts

As the names of the logical name system services state, when you use the logical name system services you will be concerned with creating, deleting, and translating logical names and creating and deleting logical name tables. There are several concepts that you should be aware of when using the logical name system services.

6.1.1 Logical Names and Equivalence Names

A *logical name* is a user-specified character string that can represent a file specification, device name, logical name table name, application-specific information, or another logical name. Typically, for process-private purposes, the user specifies logical names that are easy to use and remember. System managers and privileged users choose mnemonics for files, system devices, and search lists that are frequently accessed by all users.

Logical Name Services

Logical Name Concepts

An *equivalence name* is a character string that denotes the actual file specification, device name, or a character string. An equivalence name can also be a logical name. In this case, further translation is necessary, if permitted, to reveal the actual equivalence name.

A multivalued logical name, commonly called a *search list*, is a logical name that has more than one equivalence string. Each equivalence string is assigned an index number starting at zero.

Logical names and their equivalence strings are contained in logical name tables.

Logical names can have a maximum length of 255 characters. Equivalence strings can have a maximum of 255 characters. You can establish logical name and equivalence string pairs as follows:

- At the command level, with the DCL commands `ALLOCATE`, `ASSIGN`, `DEFINE`, or `MOUNT`
- In a program, with the Create Logical Name (`$CRELNM`), Create Mailbox and Assign Channel (`$CREMBX`), or Mount Volume (`$MOUNT`) system services

For example, you could use the symbolic name `TERMINAL` to refer to an output terminal in a program. For a particular run of the program, you could use the `DEFINE` command to establish the equivalence name `TTA2`:

To perform an assignment in a program, you must define character string descriptors for the name strings. In addition, you must call the system service through an external function declaration within your program depending on the programming language.

6.1.2 Logical Name Tables

A logical name table contains logical name and equivalence string pairs. Each table is an independent name space. Logical name tables are referenced by logical names.

Logical name tables can be created in process space or in system space. Tables created in process space are accessible only by that process. Tables created in system space are potentially shareable among many processes. Certain logical name tables have predefined logical names that provide the environment for creating, deleting, and translating user-specified logical names. These predefined logical names begin with the prefix `LNM$`. Logical name and equivalence name pairs are maintained in three types of logical name tables:

- Logical Name Directory Tables
- Default Logical Name Tables
- User-Defined Logical Name Tables

When the process is created, the logical name directory tables and the default logical name tables are created for each new process.

6.1.2.1

Logical Name Directory Tables

Because the names of logical name tables are logical names, table names must reside in logical name tables. Two special tables called directories exist for this purpose. Table names are translated from these logical name directory tables. Logical name and equivalence name pairs for logical name tables are maintained in the two directory tables shown below:

- Process Directory Table (LNM\$PROCESS_DIRECTORY)
- System Directory Table (LNM\$SYSTEM_DIRECTORY)

The process directory table contains the names of all process-private user-defined logical name tables created through the \$CRELNT system service. In addition, the process directory table contains system-assigned logical name table names, the name of the process logical name table LNM\$PROCESS_TABLE, and the default logical name table search list.

The system directory table contains the names of potentially shareable logical name tables and system-assigned logical name table names. SYSPRV privilege is required to create a logical name in the system directory table. See Section 6.1.3 for a discussion of privileges.

Logical names other than logical name table names may exist within these tables. The maximum length of logical names created in either of these tables must not exceed 31 characters. Logical names created in the directory tables must consist of alphanumeric characters, dollar signs, and underscores. Equivalence strings must not exceed 255 characters in length.

6.1.2.2

Default Logical Name Tables

Certain logical name tables are created for or assigned to a process at process creation. These tables are called the *default logical name tables*. The newly created process is provided with these tables by default. Logical name and equivalence name pairs are maintained in the default logical name tables.

Each default logical name table has a logical name associated with it. To place an entry in a logical name table, specify a logical name table name. The default logical name table names and the common logical names used to refer to them are as follows:

Table	Name	Logical Name
Process	LNM\$PROCESS_TABLE	LNM\$PROCESS
Job	LNM\$JOB_XXXXXXXX	LNM\$JOB
Group	LNM\$GROUP_gggggg	LNM\$GROUP
System	LNM\$SYSTEM_TABLE	LNM\$SYSTEM

The letter x represents a numeral in an 8-digit hexadecimal number that uniquely identifies the job logical name table. The letter g represents a numeral in a 6-digit octal number that contains the user's group number.

The maximum length of logical names created in these tables must not exceed 255 characters with no restriction on the types of characters used. Equivalence strings must not exceed 255 characters in length.

Logical Name Services

Logical Name Concepts

Process Logical Name Table

The process logical name table LNM\$PROCESS_TABLE contains names used exclusively by the process. A process logical name table is created and exists for each process in the system. Some entries in the process logical name table are made by system programs executing at more privileged access modes; these entries are qualified by the access mode from which the entry was made. The process logical name table contains the following process-permanent logical names:

Logical Name	Meaning
SYSS\$INPUT	Default input stream
SYSS\$OUTPUT	Default output stream
SYSS\$COMMAND	Original first-level (SYSS\$INPUT) input stream
SYSS\$ERROR	Default device to which the system writes error messages

SYSS\$COMMAND is only created for processes that execute LOGINOUT.

Process-Private Logical Name Creation and Image Rundown

Most entries in the process logical name table are made at user and supervisor mode. The following example shows how process-private logical names can be created in user mode by an image.

```
LOGDESC:
  .ASCID                      /ABC/
EQVNAM1:
  .ASCII                      /XYZ/
EQVLEN1=
  .-EQVNAM1
EQVNAM2:
  .ASCII                      /DEF/
EQVLEN2=
  .-EQVNAM2
TABDESC:
  .ASCID                      /LNM$PROCESS/
CRELST:
  .WORD                      EQVLEN1
  .WORD                      LNM$_STRING
  .ADDRESS                   EQVNAM1
  .LONG                      0
  .WORD                      EQVLEN2
  .WORD                      LNM$_STRING
  .ADDRESS                   EQVNAM2
  .LONG                      0
  .LONG                      0
$CRELNM_S -
  LOGNAM = LOGDESC, -          ;logical name
  TABNAM = TABDESC, -         ;table
  ITMLST = CRELST             ;equivalence strings
```

In the preceding example a logical name ABC was created and represents two equivalence strings XYZ and DEF. Each time the LNM\$_STRING item code of the *itmlst* argument is invoked, an index value is assigned to the next equivalence string. The newly created logical name and its equivalence string are contained in the process logical name table LNM\$PROCESS_TABLE.

The following example illustrates logical name creation at supervisor mode through DCL.

```
$ DEFINE/SUPERVISOR_MODE/TABLE=LNM$PROCESS ABC XYZ,DEF
```

Logical Name Services

Logical Name Concepts

Process logical names created in user mode are deleted whenever the creating process runs an image down. This behavior is illustrated by the following DCL commands:

```
# DEFINE/USER ABC XYZ
# SHOW TRANSLATION ABC
ABC = XYZ
# DIRECTORY
# SHOW LOGICAL ABC
ABC = (undefined)
```

The DCL command DIRECTORY performs image rundown when it is finished operating. At that time, all user mode process-private logical names are deleted, including the logical name ABC.

Job Logical Name Table

The job logical name table is a shareable table accessible by all processes within the same job tree. Whenever a detached process is created, a job logical name table is created for this process and all of its potential subprocesses. At the same time, the process-private logical name LNM\$JOB is created in the process directory logical name table LNM\$PROCESS_DIRECTORY. The logical name LNM\$JOB translates to the name of the job logical name table.

When a subprocess is created, only the process-private logical name LNM\$JOB is created because the job logical name table already exists for the main process.

The job logical name table contains the following three process-permanent logical names for processes that execute LOGINOUT as follows:

Logical Names	Meaning
SY\$LOGIN	Original default device and directory
SY\$LOGIN_DEVICE	Original default device
SY\$SCRATCH	Default device and directory to which temporary files are written

Thus, instead of creating these logical names within the process logical name table LNM\$PROCESS_TABLE for every process within a job tree, LOGINOUT creates these logical names once, when LOGINOUT is executed for the process at the root of the job tree.

Additionally, the job logical name table contains the following logical names:

- The logical name optionally specified and associated with a newly created temporary mailbox
- The logical name optionally specified and associated with a privately mounted volume

No privileges are required to modify the job logical name table. See Section 6.1.3 for a discussion of privileges.

Logical Name Services

Logical Name Concepts

Group Logical Name Table

The group logical name table contains names that cooperating processes in the same group can use. The GRPNAM privilege is required to add or delete a logical name in the group logical name table. See Section 6.1.3 for a discussion of privileges.

Group logical name tables are created as needed. However, the logical name LNM\$GROUP exists in each process's process directory LNM\$PROCESS_DIRECTORY. This logical name translates into the name of the group logical name table.

System Logical Name Table

The system logical name table LNM\$SYSTEM_TABLE contains names that all processes in the system can access. This table includes the default names for all system-assigned logical names. The SYSNAM or SYSPRV privilege is required to add or delete a logical name in the system logical name table. See Section 6.1.3 for a discussion of privileges.

6.1.2.3

User-Defined Logical Name Tables

You can create process-private tables and shareable tables by calling the \$CRELNT system service in a program. However, you must have SYSPRV privilege to create a shareable table. See Section 6.1.3 for a discussion of privileges.

Processes other than the creating process cannot use logical names contained in process-private tables.

Logical name tables are created through the \$CRELNT system service either with the DCL CREATE/NAME_TABLE command or by calling \$CRELNT in a program. If granted access, processes other than the creating process can use the shareable table.

The maximum length of logical names created in user-defined logical name tables must not exceed 255 characters. Equivalence strings must not exceed 255 characters in length.

6.1.3 Privileges

Certain functions of the logical name system services are restricted to users with specific privileges. The system checks the privileges in the User Authorization File (UAF) granted to you when your system manager set up your account. The system also checks for read, write, and delete accessibility. Privileges allow users to perform the following functions as shown in Table 6-1.

Logical Name Services

Logical Name Concepts

Table 6-1 Summary of Privileges

Privilege	Function
GRPNAM	Create or delete a logical name in your group logical name table.
GRPPRV	Create or delete a logical name in your group logical name table.
SYSNAM	Create executive or kernel mode logical names. Create or delete a logical name in your system logical name table. Delete a logical name or table at an inner access mode.
SYSPRV	Create or delete a logical name in your group logical name table. Create or delete a logical name in your system logical name table. Create a shareable table.

However, all users can create, delete, and translate their own process-private logical names and process-private logical name tables.

6.1.4 Attributes

Generally, attributes specified through the logical name system services perform two functions: affect the creation of logical names or govern how the system service operates and affect the translation of logical names and equivalence strings.

Attributes that affect the creation of the logical names are specified optionally in the **attr** argument of a system service call.

- **LNMSM_CONFINE**—Prevents process-private logical names from being copied to subprocesses. Subprocesses are created by the DCL SPAWN command or the LIB\$SPAWN Run-Time Library procedure. This attribute is specified only in a \$CRELNM or \$CRELNT system service call.
- **LNMSM_NO_ALIAS**—Prevents creation of a duplicate logical name in the specified logical name table at an outer access mode. If another logical name already exists in the table at an outer access mode, it is deleted.

If specified in a \$CRELNT system service call, this attribute prevents creation of a logical name table at an outer access mode in a directory table if the table name already exists in the directory table.

This attribute is specified only in a \$CRELNM or \$CRELNT system service call.

- **LNMSM_CREATE_IF**—Prevents creation of a logical name table if the specified table already exists at the specified access mode in the appropriate directory table. This attribute is specified only in a \$CRELNT system service call.
- **LNMSM_CASE_BLIND**—Governs the translation process and causes \$TRNLNM to ignore uppercase and lowercase differences in letters when searching for logical names. This attribute is specified only in a \$TRNLNM system service call.

The translation attributes **LNMSM_CONCEALED** and **LNMSM_TERMINAL** associated with logical names and equivalence strings are specified optionally through the **LNMS_ATTRIBUTES** item code in the **itmlst** argument of the \$CRELNM system service call. When the item code **LNMS_ATTRIBUTES** is specified through \$TRNLNM, the system returns the current attributes associated with the logical name and equivalence string at the current index value.

Logical Name Services

Logical Name Concepts

- **LN\$M_CONCEALED**—Indicates that the equivalence string at the current index value for the logical name is an RMS concealed device name.
- **LN\$M_CONFINE**—Indicates that the logical name cannot be used by spawned subprocesses. Subprocesses are created through the DCL SPAWN command or the Run-Time Library LIB\$SPAWN routine.
- **LN\$M_CRELOG**—Indicates the logical name was created through the \$CRELOG system service.
- **LN\$M_EXISTS**—Indicates that the equivalence string at the specified index value does exist.
- **LN\$M_NO_ALIAS**—Indicates that if the logical name already exists in the table, it cannot be created in that table at an outer access mode.
- **LN\$M_TABLE**—Indicates the logical name is the name of a logical name table.
- **LN\$M_TERMINAL**—Indicates that the equivalence strings cannot be translated further.

The attributes of multiple equivalence strings do not have to be the same. Refer to the appropriate system service in Part II of the *VAX/VMS System Services Reference Manual* for more information about attributes.

6.1.5 Logical Name Table Quotas

A logical name table *quota* is the number of bytes allocated in memory for logical names contained in a logical name table. Logical name table quotas are established in the following instances:

- Initializing the system
- Process creation
- Creating logical name tables

Each logical name table has a quota associated with it which limits the number of bytes of memory (either process pool or system paged pool) that can be occupied by the names defined in the table. The quota for a table is established when the table is created.

If no quota is specified, the newly created table has unlimited quota. Note that this table may expand to consume all available process or system memory, and that all users with write access to such a shareable table can cause the unlimited consumption of system paged pool.

6.1.5.1 Directory Table Quotas

When the system is initialized, unlimited quota is automatically established for the system directory table **LN\$SYSTEM_DIRECTORY**.

When you login on the system, unlimited quota is automatically established for the process directory table **LN\$PROCESS_DIRECTORY**.

6.1.5.2

Default Logical Name Table Quotas

The process, group, and system logical name tables have unlimited quotas.

Job Logical Name Table Quotas

Because the job logical name table is a shareable table, and no special privileges are required in order to create logical names within it, the quota allocated to this logical name table is constrained at the time the table is created. Three mechanisms exist to specify the job logical name table quota at its creation time.

For all processes that activate LOGINOUT, the quota for the job logical name is obtained from the system authorization file. This allows the quota for the job to be specified on a user-by-user basis. You can modify the job logical name table quota using a parameter with the AUTHORIZE/JTQUOTA= command.

For all processes that do not activate LOGINOUT, the quota for the job logical name table may be specified as a quota list item PQL\$_JTQUOTA in the call to the Create Process (\$CREPRC) system service. If a detached process is to be created by means of the DCL RUN/DETACHED command, then the RUN/JOB_TABLE_QUOTA command is used to specify the \$CREPRC quota list item.

For all processes that do not activate LOGINOUT, and that do not specify a PQL\$_JTQUOTA quota list item in their call to \$CREPRC, the quota for the job logical name table is taken from the dynamic system generation (SYSGEN) parameter PQL\$_DJTQUOTA. SYSGEN may be used to display both PQL\$_DJTQUOTA and PQL\$_MJTQUOTA; the default and minimum job logical name table quotas respectively.

6.1.5.3

User-Defined Logical Name Table Quotas

User-defined logical name table may be created with either an explicit limited quota or no quota limit.

The presence of user-defined logical name table quotas eliminates the necessity of a privilege, for example, SYSNAM or GRPNAM, to control consumption of paged pool when creating logical names in a shareable table.

6.1.6 Logical Name and Equivalence Name Format Conventions

The operating system uses special conventions for logical name/equivalence name assignments and translation. These conventions are generally transparent to user programs; however, you should be aware of the programming considerations involved.

If a logical name string that is presented in I/O services is preceded with an underscore character (_), the I/O services bypass logical name translation, drop the underscore character, and treat the logical name as a physical device name.

At login, the system creates default logical name table entries for process permanent files. The equivalence names for these entries (for example, SYS\$INPUT and SYS\$OUTPUT) are preceded with a four-byte header that contains the following:

Logical Name Services

Logical Name Concepts

Byte(s)	Contents
0	^X1B (Escape character)
1	^X00
2-3	VAX RMS Internal File Identifier (IFI)

This header is followed by the equivalence name string. If any of your program applications must translate system-assigned logical names, the program must be prepared to check for the existence of this header and then to use only the desired part of the equivalence string. The following program segment demonstrates how to do this.

```

ILST:
    .WORD    LNM$C_NAMLENGTH
    .WORD    LNM$STRING
    .LONG    RESSTRING
    .LONG    RESDESC
    .LONG    0
TABDESC:
    .ASCID   /LNM$FILE_DEV/      ; device/file table name
LOGDESC:
    .ASCID   /INPUT_DEVICE/      ; logical name to be translated
RESDESC:
    .LONG    LNM$C_NAMLENGTH      ; descriptor for result string
    .ADDRESS -                     ; size of result string
    .ADDRESS -                     ; address of result string
RESSTRING:
    .BLKB    LNM$C_NAMLENGTH      ; result string destination
    .
    .
    $TRNLNM_S -                   ; translate logical name
        LOGNAM=LOGDESC, -
        TABNAM=TABDESC, -
        ITMLST=ILST
    BLBC     RO,ERR                ; branch if error
    CMPW     RESSTRING, ^X001B    ; is first character an escape?
    BNEQ     1$                   ; no, continue at 1$
    SUBW     #4,RESDESC            ; yes, subtract 4 from length...
    ADDL     #4,RESDESC+4         ; and add 4 to address of string
1$:
    .
    .
    .

```

6.1.7 Specifying the Logical Name Table Search List

Logical names exist as entries within logical name tables. When a logical name is to be created, deleted, or translated, a name that designates the containing logical name table must be presented. This name possesses one or more of the following characteristics:

- It is the name of a logical name table.
- It is a logical name table name that iteratively translates in the process or system directory table to the name of a logical name table.
- It is a multivalued logical name that iteratively translates to the names of several logical name tables. A multivalued logical name is also known as a search list. The tables are used in the order in which they appear.

Logical Name Services

Logical Name Concepts

As mentioned earlier, predefined logical names exist for certain logical name tables. These predefined names begin with the prefix LNM\$. You can redefine these names to modify the search order or the tables used.

Instead of a fixed set of logical name tables and a rigidly defined order (process, job, group, system) for searching those tables, you can specify the tables to be searched and the order in which they are to be searched. Logical names in the directory tables are used to specify this searching order. By convention, each class of logical name use, for example, device/file specification, uses a particular predefined name for this purpose.

For example, LNM\$FILE_DEV is the name of the logical name table used whenever file specifications or device names are translated by RMS or the I/O services. This name must translate to a list of one or more logical name table names specifying the tables to be searched when translating file specifications.

By default, LNM\$FILE_DEV specifies that the process, job, group, and system tables are all searched, in that order, and the first match found is returned.

Logical name table names are translated from two tables, the process logical name directory table LNM\$PROCESS_DIRECTORY and the system logical name directory table LNM\$SYSTEM_DIRECTORY. LNM\$FILE_DEV must be defined in one of these tables.

Thus, if identical logical names exist in the process and group tables, the process table entry is found first, and the job and group tables are not searched. When the process logical name table is searched, the entries are searched in order of access mode, with user mode entries matched first, supervisor second, and so on.

If you wish to change the list of tables used for device and file specifications, you can redefine LNM\$FILE_DEV in the process directory table LNM\$PROCESS_DIRECTORY.

6.2

Creating a Logical Name—\$CRELNM

To perform an assignment in a program, you must provide character string descriptors for the name strings, select the table to contain the logical name, and use the \$CRELNM system service as shown in the following example. In either case, the result is the same: the logical name TTY is equated to the physical device name TTA2: in table LNM\$JOB.

Logical Name Services

Creating a Logical Name—\$CRELNM

```

LOGDESC:
  .ASCID                      /TTY/
TABDESC:
  .ASCID                      /LNM$JOB/
LNMATTR:
  .LONG                       LNM$M_TERMINAL
CRELST:
  .WORD                       4
  .WORD                       LNM$ _ATTRIBUTES
  .ADDRESS                    LNMATTR
  .LONG                       0
  .WORD                       EQVLEN
  .WORD                       LNM$ _STRING
  .ADDRESS                    EQVNAM
  .LONG                       0
  .LONG                       0
EQVNAM:
  .ASCII                      /TTA2:/
EQVLEN=
  .-EQVNAM

$CRELNM_S -
  LOGNAM = LOGDESC, -
  TABNAM = TABDESC, -
  ATTR   = TABDESC, -
  ITMLST = CRELST

```

Note that the translation attribute is specified as terminal. This attribute indicates that iterative translation of the logical name TTY ends when the equivalence string TTA2: is returned. In addition, because the **acmode** argument was not specified, the access mode of the logical name TTY is the access mode of the calling image.

6.2.1 Duplication of Logical Names

A logical name table can contain entries for the same logical name at different access modes. Different logical name tables can contain entries for the same logical name.

In all other cases, there can be only one entry for a particular logical name in a logical name table.

Any number of logical names can have the same equivalence name.

Consider the following examples of the logical name **TERMINAL** defined in several tables. The logical name **TERMINAL** translates differently depending on the table that is specified.

Process Logical Name Table for Process A

The process logical name table shown below equates the logical name **TERMINAL** to the specific terminal **TTA2:.** **INFILE** and **OUTFILE** are equated to disk specifications. The logical names were created from supervisor mode.

Logical Name Services

Creating a Logical Name—\$CRELNM

Logical Name		Equivalence Name	Access Mode
INFILE	——>	DM1:[HIGGINS]TEST.DAT	Supervisor
OUTFILE	——>	DM1:[HIGGINS]TEST.OUT	Supervisor
TERMINAL	——>	TTA2:	Supervisor
.		.	.
.		.	.
.		.	.

To determine the equivalence string for the logical name TERMINAL in the preceding table, issue the following command:

SHOW LOGICAL TERMINAL

The system returns the equivalence string TTA2:.

Job Logical Name Table

The portion of the job logical name table shown below defines the logical TERMINAL to a virtual terminal VTA14:. The logical name SYS\$LOGIN is the device and directory for the process at login. SYS\$LOGIN is defined in exec mode.

Logical Name		Equivalence Name	Access Mode
SYS\$LOGIN	——>	DBA9:[HIGGINS]	Exec
TERMINAL	——>	VTA14:	User
.		.	.
.		.	.
.		.	.

To determine the equivalence string of the logical name TERMINAL defined in the preceding table, issue the following command:

SHOW LOGICAL/JOB TERMINAL

The system returns the equivalence string VTA14: as the translation.

User-Defined Logical Name Table

The user-defined logical name table LOG_TBL shown below contains a definition of TERMINAL as the mailbox device MBA407:. The multivalued logical name XYZ has two translations DISK1: and DISK2:.

Logical Name		Equivalence Name	Access Mode
TERMINAL	——>	MBA407:	Supervisor
XYZ	——>	DISK1:	Supervisor
	——>	DISK3:	
.		.	.
.		.	.
.		.	.

To determine the equivalence string for the logical name TERMINAL in the preceding user-defined table, issue the following command:

SHOW LOGICAL/TABLE=LOG_TBL TERMINAL

Logical Name Services

Creating a Logical Name—\$CRELNM

The system returns the equivalence string MBA407:. In order to use this definition of `TERMINAL` as a device or file specification, the logical name table name `LN$FILE_DEV` must be redefined to reference the user-defined table as follows:

```
* DEFINE/TABLE=LN$PROCESS_DIRECTORY LN$FILE_DEV LOG_TBL, LN$PROCESS,  
LN$JOB, LN$SYSTEM
```

In the preceding example, the DCL `DEFINE` command is used to redefine the default search list `LN$FILE_DEV`. The `/TABLE` qualifier specifies the table `LN$PROCESS_DIRECTORY` that will contain the redefined search list. The system will search the tables defined by `LN$FILE_DEV` in the following order: `LOG_TBL`, `LN$PROCESS_TABLE`, `LN$JOB`, and `LN$SYSTEM_TABLE`.

System Logical Name Table

The system logical table contains system-assigned logical names accessible to all processes in the system. For example, the logical names `SY$LIBRARY` and `SY$SYSTEM` provide logical names that all users can access to use the device and directory containing system files.

Logical Name		Equivalence Name
SY\$LIBRARY	—>	SY\$SYSROOT:[SYSLIB]
SY\$SYSTEM	—>	SY\$SYSROOT:[SYSEXE]
.	.	.
.	.	.
.	.	.

The "Logical Names" section of the *VAX/VMS DCL Dictionary* contains a list of these system-assigned logical names.

6.2.1.1

Logical Name Supersession

If the logical name `TERMINAL` is equated to `TTA2:` in the process table as shown in the previous examples, and the process subsequently equates the logical name `TERMINAL` to `TTA3:`, the equivalence of `TERMINAL TTA2:` is replaced by the new equivalence name. The successful return status code `SS$_SUPERSEDE` indicates that a new entry replaced an old one.

The definitions of `TERMINAL` in the job table and the user-defined table `LOG_TBL` are unaffected.

6.3 Creating Logical Name Tables—\$CRELNT

The `$CRELNT` system service creates logical name tables. Logical name tables can be created at any access mode depending on the privileges of the calling process. A user-specified logical name identifying the newly created logical name table is stored in the process directory table `LN$PROCESS_DIRECTORY`.

6.3.1 Shareable and Named Shareable Logical Name Tables

If you have SYSPRV privilege, you can create shareable logical name tables. You can assign protection to these tables through the **promsk** argument of the \$CRELNT system service. The **promsk** argument allows you to specify the type of access for system, owner, group, and world users as follows:

- Read privileges allow access to names in the logical name table.
- Write privileges allow creation and deletion of names within the logical name table.
- Delete privileges allow deletion of the logical name table.

Note: The "E" protection bit is reserved for DIGITAL use.

If the **promsk** argument is omitted, complete access is granted to system and owner, and no access is granted to group and world.

6.3.2 \$CRELNT System Service Call

The following example illustrates a call to the \$CRELNT system service.

```
TABDESC:      .ASCID      /LOG_TABLE/
PARDESC:      .ASCID      /LNM$PROCESS_DIRECTORY/
TAB_ATTR:     .LONG       LNM$M_CONFINE
TAB_QUOTA:    .LONG       5000
.
.
$CRELNT_S -
    TABNAM = TABDESC,-      ;table name
    PARTAB = PARDESC,-      ;parent table
    ATTR = TAB_ATTR,-       ;attributes
    QUOTA = TAB_QUOTA       ;quota
```

In the preceding example, a user-defined table LOG_TABLE is created with an explicit quota of 5000 bytes. The name of the newly created table is an entry in the parent table LNM\$PROCESS_DIRECTORY. Since the CONFINE attribute is associated with the logical name table, the table cannot be copied from the process to its spawned processes.

6.4 Deleting Logical Names—\$DELLNM

The \$DELLNM system service deletes entries from a logical name table. When you write a call to the \$DELLNM system service, you can specify a single logical name to delete, or you can specify that you want to delete all logical names from a particular table. For example, the following call deletes the process logical name TERMINAL from the job logical name table.

```
LOGDESC:      .ASCID      /TERMINAL/
TABDESC:      .ASCID      /LNM$JOB/
$DELLNM_S -
    LOGNAM = LOGDESC,-
    TABNAM = TABDESC,-
```


Logical Name Services

Deleting Logical Names—\$DELLNM

Logical names or logical name table names that were placed in a process-private logical name table from an image running in user mode and the DCL commands ASSIGN/USER and DEFINE/USER are automatically deleted at image exit. Entries made from the command stream are placed in the table by the command interpreter; these are supervisor mode entries and are not deleted at image exit (except for the logical names defined by the DCL commands ASSIGN/USER and DEFINE/USER).

6.5

Translating Logical Names—\$TRNLNM

The \$TRNLNM system service translates a logical name to its equivalence string. In addition, \$TRNLNM returns information about the logical name and equivalence string.

The tables in which to search for the logical name are specified in the system service call to \$TRNLNM. This table argument can be either the name of a logical name table or it can be a logical name that translates to a list of one or more logical name tables.

Since logical names can have many equivalence strings, you can specify which equivalence string you wish to receive.

A number of system services that require a device name accept a logical name and translate the logical name iteratively until a physical device name is found (or until the system default number of logical name translations has been performed). These services implicitly specify the logical name table name LNM\$FILE_DEV. Refer to "Specifying the Logical Name Table Search List" earlier in this section for more information about LNM\$FILE_DEV. The following system services perform iterative logical name translation automatically:

- Allocate Device (\$ALLOC)
- Assign I/O Channel (\$ASSIGN)
- Broadcast (\$BRDCST)
- Create Mailbox (\$CREMBX)
- Deallocate Device (\$DALLOC)
- Dismount Volume (\$DISMOU)
- Get Device/Volume Information (\$GETDVI)
- Mount Volume (\$MOUNT)

In many cases, however, a program must perform the logical name translation to obtain the equivalence name for a logical name outside the context of a device name or file specification. In that case, you must supply the name of the table or tables to be searched. The Translate Logical Name (\$TRNLNM) system service searches the user-specified logical name tables for a specified logical name and returns the equivalence name. In addition, \$TRNLNM returns attributes specified optionally for the logical name and equivalence string.

The following example shows a call to the \$TRNLNM system service to translate the logical name ABC.

Logical Name Services

Translating Logical Names—\$TRNLNM

```
LOGDESC: .ASCID /ABC/
TABDESC: .ASCID /LNM$FILE_DEV/
EQVBUF1: .BLKB LNM$C_NAMLENGTH
EQVDESC1: .LONG 0
           .ADDRESS EQVBUF1
EQVBUF2: .BLKB LNM$C_NAMLENGTH
EQVDESC2: .LONG 0
           .ADDRESS EQVBUF2
TRNLST:  .WORD LNM$C_NAMLENGTH
           .WORD LNM$ _STRING
           .ADDRESS EQVBUF1
           .ADDRESS EQVDESC1
           .WORD LNM$C_NAMLENGTH
           .WORD LNM$ _STRING
           .ADDRESS EQVBUF2
           .ADDRESS EQVDESC2
           .LONG 0
TRNATTR: .LONG LNM$M_CASE_BLIND
.
.
$TRNLNM_S -
LOGNAM = LOGDESC, -
TABNAM = TABDESC, -
ATTR = TRNATTR, -
ITMLST = TRNLST
```

This call to the \$TRNLNM system service results in the translation of the logical name ABC. In addition, LNM\$FILE_DEV is specified in the **tabnam** argument as the search list that \$TRNLNM will use to find logical name ABC. The logical name ABC was assigned two equivalence strings. The LNM\$_STRING item code in the **itmlst** argument directs \$TRNLNM to look for an equivalence string at the current index value. Note that the LNM\$_STRING item code is invoked twice. The equivalence strings are placed in the two output buffers EQVBUF1 and EQVBUF2 described by TRNLST.

The attribute LNM\$M_CASE_BLIND governs the translation process. \$TRNLNM searches for the equivalence strings without regard to uppercase or lowercase letters. \$TRNLNM matches any of the following character strings: ABC, aBC, AbC, abc, and so forth.

The output equivalence name string length is written into the first word of the character string descriptor. This descriptor can then be used as input to another system service.

6.6 Example of Using the Logical Name System Services

```

                                CALC.FOR
PROGRAM CALC
Include '($lnmdef)'
! status variable and system routines
INTEGER*4 STATUS,
2      SYS$CRELNM,
2      LIB$GET_EF,
2      LIB$SPAWN
! item list for SYS$CRELNM
INTEGER*2 NAME_LEN,
2      NAME_CODE
```

Logical Name Services

Example of Using the Logical Name System Services

```

INTEGER*4 NAME_ADDR,
2      RET_ADDR /0/,
2      END_LIST /0/
COMMON /LIST/ NAME_LEN,
2      NAME_CODE,
2      NAME_ADDR,
2      RET_ADDR,
2      END_LIST
! number to pass to REPEAT.FOR
CHARACTER*3 REPETITIONS_STR
INTEGER REPETITIONS
! symbols for LIB$SPAWN and SYS$CRELNM
EXTERNAL CLI$M_NOLOGNAM,
2      CLI$M_NOCLISYM,
2      CLI$M_NOKEYPAD,
2      CLI$M_NOWAIT,
2      LNM$STRING
. ! set REPETITIONS_STR

! set up and create logical name REP_NUMBER in job table
NAME_LEN = 3
NAME_CODE = (LNM$STRING)
NAME_ADDR = %LOC(REPETITIONS_STR)
STATUS = SYS$CRELNM (,
2      'LNM$JOB', ! logical name table
2      'REP_NUMBER', ! logical name
2      NAME_LEN) ! list specifying
! equivalence string
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
! execute REPEAT.FOR in a subprocess
MASK = %LOC (CLI$M_NOLOGNAM) .OR.
2      %LOC (CLI$M_NOCLISYM) .OR.
2      %LOC (CLI$M_NOKEYPAD) .OR.
2      %LOC (CLI$M_NOWAIT)
STATUS = LIB$GET_EF (FLAG)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
STATUS = LIB$SPAWN ('RUN REPEAT'...,MASK,,,FLAG)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
.
.
.
REPEAT.FOR
PROGRAM REPEAT
! Repeats a calculation REP_NUMBER of times,
! where REP_NUMBER is a symbol name
! status variables and system routines
INTEGER STATUS,
2      SYS$TRNLNM,SYS$DELLNM
! number of times to repeat
INTEGER*4 REITERATE,
2      REPEAT_STR_LEN
CHARACTER*3 REPEAT_STR
! item list for SYS$TRNLNM
INTEGER*2 NAME_LEN,
2      NAME_CODE
INTEGER*4 NAME_ADR,
2      RET_ADR,
2      END_LIST /0/
COMMON /LIST/ NAME_LEN,
2      NAME_CODE,
2      NAME_ADDR,
2      RET_ADDR,
2      END_LIST
EXTERNAL LNM$STRING
! set up and translate the logical name REP_NUMBER
NAME_LEN = 3
NAME_CODE = (LNM$STRING)
NAME_ADDR = %LOC(REPEAT_STR)

```

Logical Name Services

Example of Using the Logical Name System Services

```

RET_ADDR = %LOC(REPEAT_STR_LEN)
STATUS = SYS$STRNLNM (,
2      'LNM$JOB',      ! logical name table
2      'REP_NUMBER',.. ! logical name
2      NAME_LEN)      ! list requesting
                        ! equivalence string
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
! convert equivalence string to integer
READ (UNIT = REPEAT_STR (1:REPEAT_STR_LEN),
2     FMT = '(I3)') REITERATE
! calculations
DO I = 1, REITERATE
.
.
.
END DO
! delete logical name
STATUS = SYS$DELLNM ('LNM$JOB',      ! logical name table
2      'REP_NUMBER',) ! logical name
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
END

```

In the above example, the FORTRAN program CALC.FOR creates a spawned subprocess to perform an iterative calculation. The logical name REP_NUMBER specifies the number of times that REPEAT should perform the calculation. Since the two processes are part of the same job, REP_NUMBER is placed in the job logical name table LNM\$JOB. (Note that logical name table names are case sensitive; specifically, LNM\$JOB is a system-defined logical name that refers to the job logical name table, lnm\$job is not.)

You can use two basic methods to perform input/output operations under VAX/VMS.

- VAX Record Management Services (RMS)
- I/O system services

VAX RMS provides a set of routines for general purpose, device-independent functions such as data storage, retrieval, and modification.

The I/O system services permit you to use the I/O resources of the operating system directly in a device-dependent manner. I/O services also provide some specialized functions not available in VAX RMS. Using I/O services requires more knowledge on your part than if you used VAX RMS, but can result in more efficient input/output operations.

The system services listed below are Input/Output Services.

- Assign I/O Channel (\$ASSIGN)
- Deassign I/O Channel (\$DASSGN)
- Queue I/O Request (\$QIO)
- Queue I/O Request and Wait for Event Flag (\$QIOW)
- Formatted ASCII Output (\$FAO)
- Formatted ASCII Output with List Parameter (\$FAOL)
- Allocate Device (\$ALLOC)
- Deallocate Device (\$DALLOC)
- Mount Volume (\$MOUNT)
- Dismount Volume (\$DISMOU)
- Get Device and Channel Information (\$GETDVI)
- Get Device and Channel Information and Wait (\$GETDVIW)
- Cancel I/O on Channel (\$CANCEL)
- Create Mailbox and Assign Channel (\$CREMBX)
- Delete Mailbox (\$DELMBX)
- Breakthrough (\$BRKTH)
- Breakthrough and Wait (\$BRKTHW)
- Send Message to Job Controller (\$SNDJBC)
- Send Message to Job Controller and Wait (\$SNDJBCW)
- Send Message to Operator (\$SNDOPR)
- Send Message to Error Logger (\$SNDERR)

Input/Output Services

- Get Message (\$GETMSG)
- Put Message (\$PUTMSG)

This section provides general information on how to use the I/O services, including:

- Assigning channels
- Queuing I/O requests
- Allocating devices
- Using mailboxes

Examples are provided to show you how to use the I/O services for simple functions, such as terminal input and output operations. If you plan to write device-dependent I/O routines, see the *VAX/VMS I/O Reference Volume*.

The following methods of performing I/O with VAX/VMS are documented in *Writing a Device Driver for VAX/VMS*.

- Writing your own device driver.
- Connecting to a device interrupt vector.

7.1 Quotas, Privileges, and Protection

To preserve the integrity of the system, VAX/VMS I/O operations are performed under the constraints of quotas, privileges, and protection.

Quotas establish a limit on the number and type of I/O operations that a process can perform concurrently, and on the total size of outstanding transfers. They ensure that all users have an equitable share of system resources and usage.

Privileges are granted to a user to allow the performance of certain I/O-related operations, for example, creating a mailbox and performing logical I/O to a file-structured device. Restrictions on user privilege protect the integrity and performance of both the operating system and the services provided to other users.

Protection is used to control access to files and devices. Device protection is provided in much the same way as file protection: shareable and nonshareable devices are protected by protection masks.

The Set Resource Wait Mode (\$SETRWM) system service allows a process to select either of two modes when an attempt to exceed a quota occurs. In the enabled (default) mode, the process waits until the required resource is available before continuing. In the disabled mode, the process is notified immediately by a system service status return that an attempt to exceed a quota has occurred. Waiting for resources is transparent to the process when resource wait mode is enabled; no explicit action is taken by the process when a wait is necessary.

The different types of I/O-related quotas, privileges, and protection are described in the following sections.

7.1.1 Buffered I/O Quota

The buffered I/O quota specifies the maximum number of concurrent buffered I/O operations a process can have active. In a buffered I/O operation, the user's data is buffered in system dynamic memory. The driver deals with the system buffer and not the user buffer. Buffered I/O is used for terminal, line printer, card reader, network, mailbox, and console medium (RX01) transfers and ACP operations. The user's buffer does not have to be locked in memory for a buffered I/O operation.

The buffered I/O quota value is established in the user authorization file by the system manager or by the process's creator. Resource wait mode is entered if enabled by the Set Resource Wait Mode system service and an attempt to exceed the buffered I/O quota is made.

7.1.2 Buffered I/O Byte Count Quota

The buffered I/O byte count quota specifies the maximum amount of buffer space that can be consumed from system dynamic memory for buffering I/O requests. All buffered I/O requests require system dynamic memory in which the actual I/O operation takes place.

The buffered I/O byte count quota is established in the user authorization file by the system manager or by the process's creator. Resource wait mode is entered if enabled by the Set Resource Wait Mode system service and an attempt to exceed the buffered I/O byte count quota is made.

7.1.3 Direct I/O Quota

The direct I/O quota specifies the maximum number of concurrent direct (that is, unbuffered) I/O operations that a process can have active. In a direct I/O operation, data is moved directly to or from the user buffer. Direct I/O is used for disk, magnetic tape, most DMA real-time devices, and nonnetwork transfers, for example, DMC11/DMR11 write transfers. For direct I/O, the user's buffer must be locked in memory during the transfer.

The direct I/O quota value is established in the user authorization file by the system manager or by the process's creator. If you use the Set Resource Wait Mode system service to enable resource wait mode for the process, the process enters resource wait mode if it attempts to exceed its direct I/O quota value.

7.1.4 AST Quota

The AST quota specifies the maximum number of asynchronous system traps that a process can have outstanding. The quota value is established in the user authorization file by the system manager or by the process's creator. There is never an implied wait for that resource.

Input/Output Services

Quotas, Privileges, and Protection

7.1.5 Physical I/O Privilege

Physical I/O privilege (PHY_IO) allows a process to perform physical I/O operations on a device. Physical I/O privilege also allows a process to perform logical I/O operations on a device. Figures 7-4 (Physical I/O Access Checks) and 7-5 (Logical I/O Access Checks) show the use of physical I/O privilege in greater detail.

7.1.6 Logical I/O Privilege

Logical I/O privilege (LOG_IO) allows a process to perform logical I/O operations on a device. A process can also perform physical operations on a device if the process has logical I/O privilege, the volume is mounted foreign, and the volume protection mask allows access to the device. (A foreign volume is a volume that does not contain a standard file structure understood by any VAX/VMS software.) Figures 7-4 (Physical I/O Access Checks) and 7-5 (Logical I/O Access Checks) show the use of logical I/O privilege in greater detail.

7.1.7 Mount Privilege

Mount privilege (MOUNT) allows a process to use the IO\$_MOUNT function to perform mount operations on disk and magnetic tape devices. IO\$_MOUNT is used in ACP interface operations.

7.1.8 Volume Protection

Volume protection protects the integrity of mailboxes and both foreign and Files-11 structured volumes. Volume protection for a foreign volume is established when the volume is mounted. Volume protection for a Files-11 structured volume is established when the volume is initialized. (The protection can be overridden when the volume is mounted if the process that is mounting the volume has the override volume protection privilege, VOLPRO.)

Mailbox protection is established by the \$CREMBX system service protection mask argument.

Set Protection QIO requests allow the user to set volume protection on a mailbox. The requester must either be the owner of the mailbox or have BYPASS privilege.

Protection for structured volumes and mailboxes is provided by a volume protection mask that contains four 4-bit fields. These fields correspond to the four classes of users that are permitted to access the volume. (User classes are based on the volume owner's UIC.)

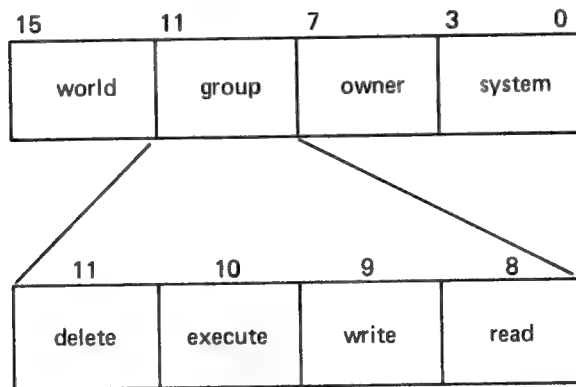
The 4-bit fields are interpreted differently for volumes that are mounted as structured (that is, volumes serviced by an ancillary control process (ACP)), volumes that are mounted as foreign, and mailboxes (both temporary and permanent).

Figure 7-1 shows the 4-bit protection fields for volumes mounted as structured.

Input/Output Services

Quotas, Privileges, and Protection

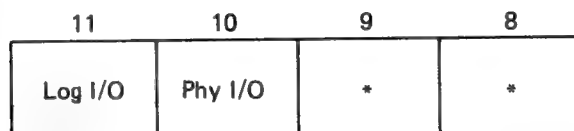
Figure 7-1 Files-11 Volume Protection Fields



ZK-622-82

Figure 7-2 shows the 4-bit protection fields for foreign volumes.

Figure 7-2 Foreign Volume Protection Fields

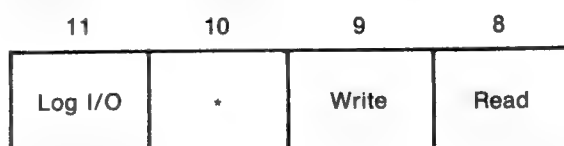


*not used

ZK-623-82

Figure 7-3 shows the 4-bit protection fields for mailboxes.

Figure 7-3 Mailbox Protection Fields



*not used

ZK-624-82

Usually, volume protection is meaningful only for read and write operations.

Input/Output Services

Quotas, Privileges, and Protection

7.1.9 Device Protection

Device protection protects the allocation of nonshareable devices, such as terminals and card readers.

Protection is provided by a device protection mask similar to that of volume protection. The difference is that only the bit corresponding to read access is checked and that bit determines if the process can allocate or assign a channel to the device.

Device protection is established with the DCL command SET PROTECTION /DEVICE. Both the protection mask and the device owner UIC are set with this command.

7.1.10 System Privilege

System UIC privilege (SYSPRV) allows a process to be eligible for the volume or device protection specified for the system protection class, even though the process does not have a UIC in one of the system groups.

7.1.11 Bypass Privilege

Bypass privilege (BYPASS) allows a process to completely bypass volume and device protection.

7.2 Summary of VAX/VMS QIO Operations

The VAX/VMS system provides QIO operations that perform three basic I/O functions: read, write, and set mode. The read function transfers data from a device to a user-specified buffer. The write function transfers data in the opposite direction—from a user-specified buffer to the device. For example, in a read QIO function to a terminal device, a user-specified buffer is filled with characters received from the terminal. In a write QIO function to the terminal, the data in a user-specified buffer is transferred to the terminal where it is displayed.

The set mode QIO function is used to control or describe the characteristics and operation of a device. For example, a set mode QIO function to a line printer can specify either uppercase or lowercase character format. Not all QIO functions are applicable to all types of devices. The line printer, for example, cannot perform a read QIO function.

7.3 Physical, Logical, and Virtual I/O

I/O data transfers can occur in any one of three device addressing modes: physical, logical, or virtual. Any process with device access allowed by the volume protection mask can perform logical I/O on a device that is mounted foreign; physical I/O requires privilege. Virtual I/O does not require privilege; however, intervention by an ACP to control user access may be necessary if the device is under ACP control. (ACP functions are described in the *VAX/VMS I/O Reference Volume*.)

7.3.1 Physical I/O Operations

In physical I/O operations, data is read from and written to the actual, physically addressable units accepted by the hardware (for example, sectors on a disk or binary characters on a terminal in the PASSALL mode). This mode allows direct access to all device-level I/O operations.

Physical I/O requires that one of the following conditions be met:

- The issuing process has physical I/O privilege (PHY_IO).
- The issuing process has all of the following characteristics:
 - The issuing process has logical I/O privilege (LOG_IO).
 - The device is mounted foreign.
 - The volume protection mask allows physical access to the device.

If neither of these conditions is met, the physical I/O operation is rejected by the \$QIO system service, which returns a condition value of SS\$_NOPRIV (no privilege). Figure 7-4 illustrates the physical I/O access checks in greater detail.

The inhibit error-logging function modifier (IO\$_M_INHERLOG) can be specified for all physical I/O functions. IO\$_M_INHERLOG inhibits the logging of any error that occurs during the I/O operation.

7.3.2 Logical I/O Operations

In logical I/O operations, data is read from and written to logically addressable units of the device. Logical operations can be performed on both block-addressable and record-oriented devices. For block-addressable devices (such as disks), the addressable units are 512-byte blocks. They are numbered from 0 to $n-1$ where n is the number of blocks on the device. For record-oriented or non-block-structured devices (such as terminals), logical addressable units are not pertinent and are ignored. Logical I/O requires that one of the following conditions be met:

- The issuing process has physical I/O privilege (PHY_IO).
- The issuing process has logical I/O privilege (LOG_IO).
- The volume is mounted foreign and the volume protection mask allows access to the device.

If none of these conditions is met, the logical I/O operation is rejected by the \$QIO system service, which returns a condition value of SS\$_NOPRIV (no privilege). Figure 7-5 illustrates the logical I/O access checks in greater detail.

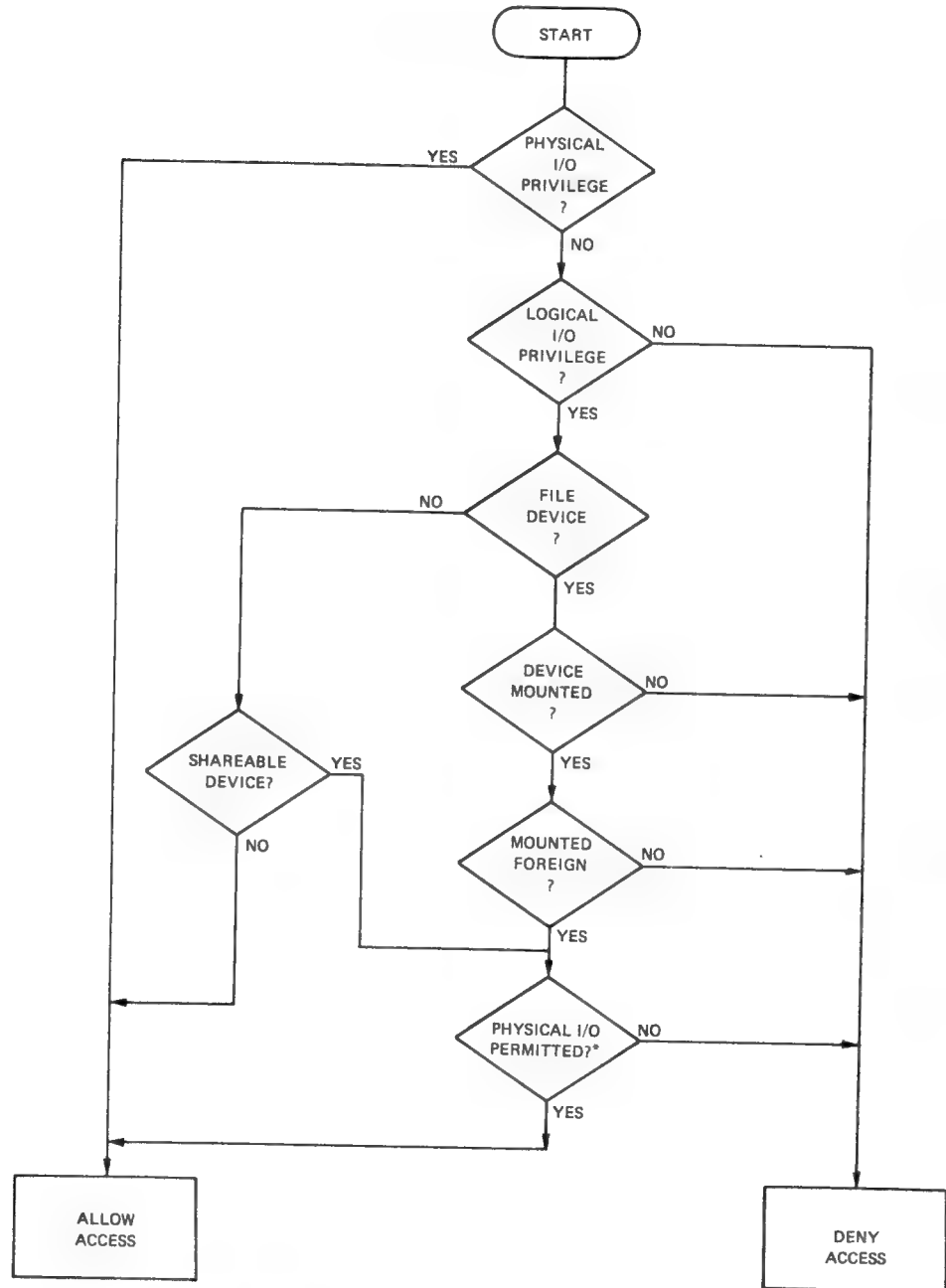
7.3.3 Virtual I/O Operations

Virtual I/O operations can be performed on both record-oriented (non-file-structured) and block-addressable (file-structured) devices. For record-oriented devices (such as terminals), the virtual function is the same as a logical function; the virtual addressable units of the devices are ignored.

Input/Output Services

Physical, Logical, and Virtual I/O

Figure 7-4 Physical I/O Access Checks



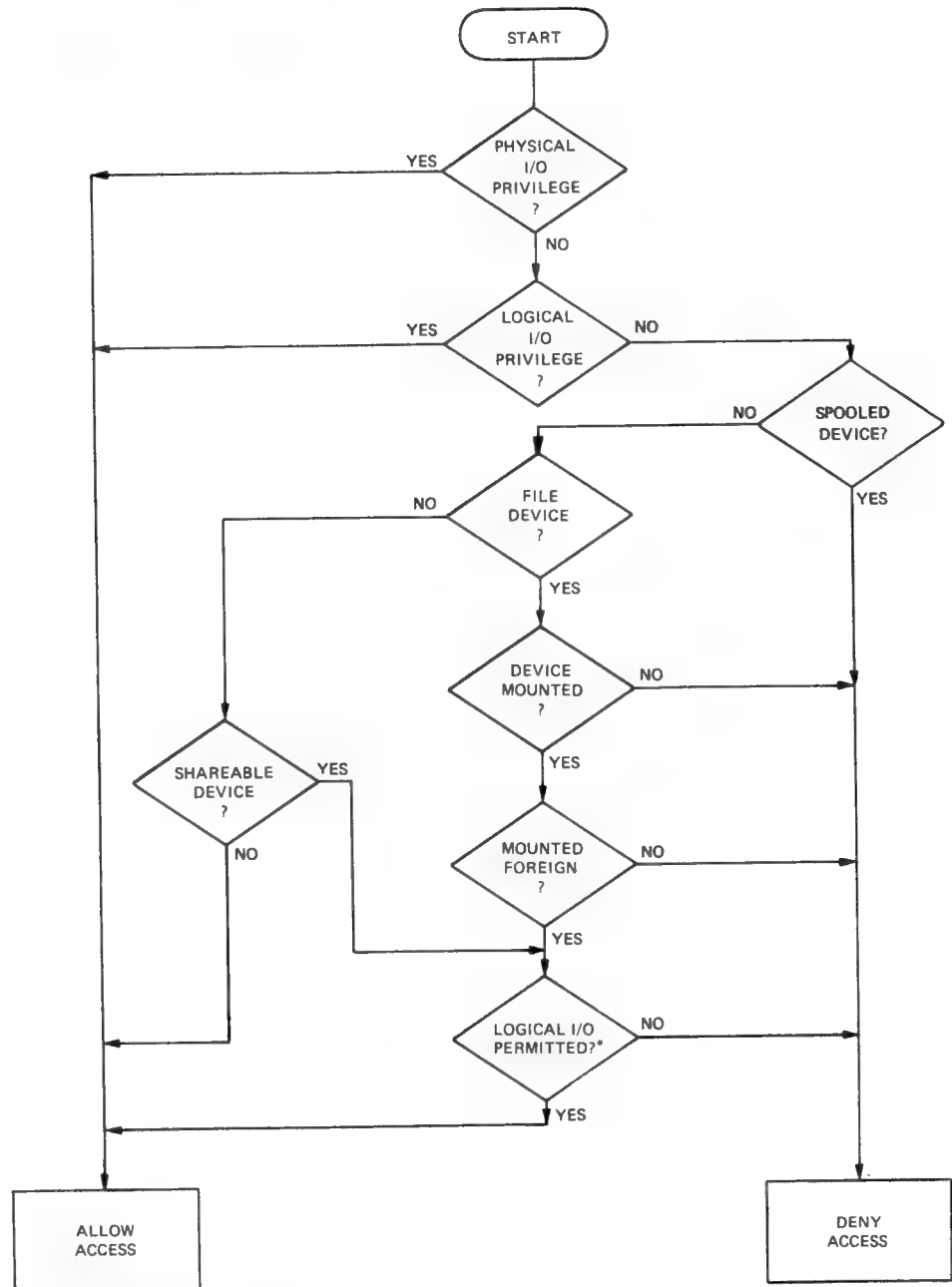
*Volume protection mask allows access

ZK-825-82

Input/Output Services

Physical, Logical, and Virtual I/O

Figure 7-5 Logical I/O Access Checks



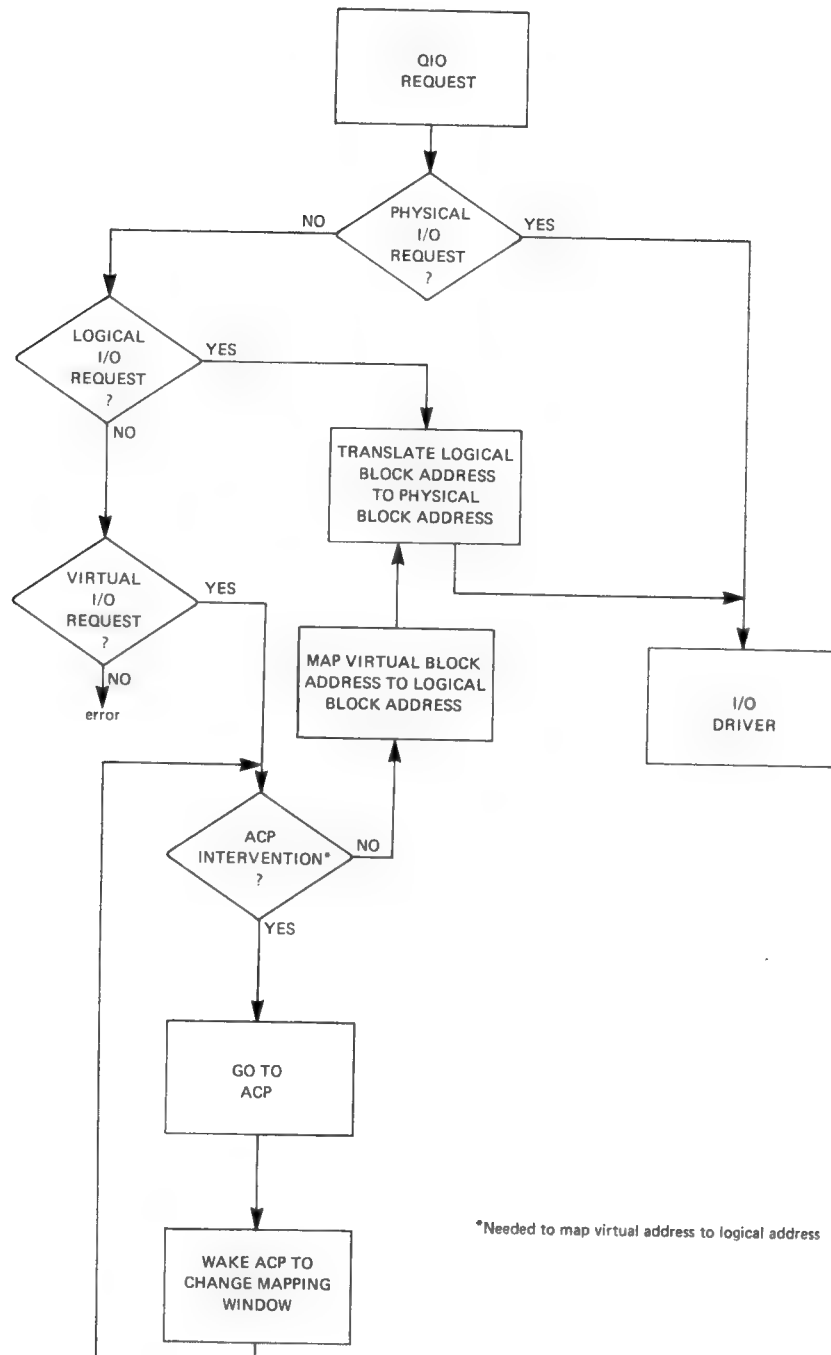
*Volume protection mask allows access

ZK-826-82

Input/Output Services

Physical, Logical, and Virtual I/O

Figure 7-6 Physical, Logical, and Virtual I/O



ZK-627-82

Input/Output Services

I/O Function Encoding

For block-addressable devices (such as disks), data is read from and written to open files. The addressable units in the file are 512-byte blocks. They are numbered starting at 1 and are relative to a file rather than to a device. Block-addressable devices must be mounted and structured and must contain a file that was previously accessed on the I/O channel.

Virtual I/O operations also require that the volume protection mask allow access to the device (a process having either physical or logical I/O privilege can override the volume protection mask). If these conditions are not met, the virtual I/O operation is rejected by the QIO system service, which returns one of the following condition values.

Condition Value	Meaning
SS\$_NOPRIV	No privilege
SS\$_DEVNOTMOUNT	Device not mounted
SS\$_DEVFOREIGN	Volume mounted foreign

Figure 7-6 shows the relationship of physical, logical, and virtual I/O to the driver.

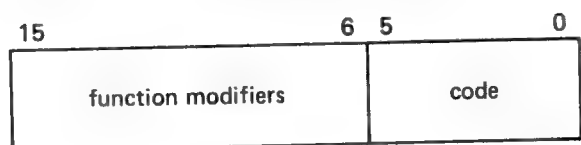
7.4 I/O Function Encoding

I/O functions fall into three groups that correspond to the three I/O device addressing modes (physical, logical, and virtual) described in Section 7.3. Depending on the device to which it is directed, an I/O function can be expressed in one, two, or all three modes.

I/O functions are described by 16-bit, symbolically expressed values that specify the particular I/O operation to be performed and any optional function modifiers. Figure 7-7 shows the format of the 16-bit function value.

Symbolic names for I/O function codes are defined by the \$IODEF macro.

Figure 7-7 I/O Function Format



ZK-628-82

7.4.1 Function Codes

The low-order 6 bits of the function value are a code that specifies the particular operation to be performed. For example, the code for read logical block is expressed as IO\$_READLBLK. Table 7-1 lists the symbolic values for read and write I/O functions in the three transfer modes.

Input/Output Services

I/O Function Encoding

Table 7-1 Read and Write I/O Functions

Physical I/O	Logical I/O	Virtual I/O
IO\$_READPBLK	IO\$_READLBLK	IO\$_READVBLK
IO\$_WRITEPBLK	IO\$_WRITELBLK	IO\$_WRITEVBLK

The set mode I/O function has a symbolic value of IO\$_SETMODE.

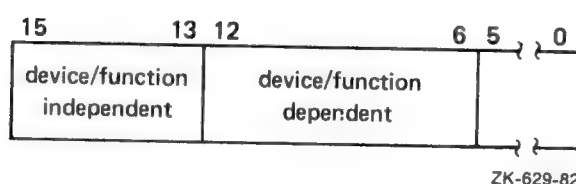
Function codes are defined for all supported devices. Although some of the function codes (for example, IO\$_READVBLK and IO\$_WRITEVBLK) are used with several types of devices, most are device dependent; that is, they perform functions specific to particular types of devices. For example, IO\$_CREATE is a device-dependent function code; it is used only with file-structured devices such as disks and magnetic tapes. The *VAX/VMS I/O Reference Volume* provides complete descriptions of the functions and function codes.

Note: Users should determine the device class before performing any QIO function, because the requested function may be incompatible with some devices. For example, the SYS\$INPUT device could be a terminal, a disk, or some other device. Unless this device is a terminal, an IO\$_SETMODE request that enables a CTRL/C AST would not be performed.

7.4.2 Function Modifiers

The high-order 10 bits of the function value are function modifiers. These are individual bits that alter the basic operation to be performed. For example, the function modifier IO\$_M_NOECHO can be specified with the function IO\$_READLBLK to a terminal. When used together, the two values are written in VAX MACRO as IO\$_READLBLK!IO\$_M_NOECHO. This causes data typed at the terminal keyboard to be entered into the user buffer, but not echoed to the terminal. Figure 7-8 shows the format of function modifiers.

Figure 7-8 Function Modifier Format



As shown, bits 13 through 15 are device/function-independent bits, and bits 6 through 12 are device/function-dependent bits. Device/function-dependent bits have the same meaning, whenever possible, for different device classes. For example, the function modifier IO\$_M_ACCESS is used with both disk and magnetic tape devices to cause a file to be accessed during a create operation. Device/function-dependent bits always have the same function within the same device class.

There are two device/function-independent modifier bits: IO\$_M_INHRETRY and IO\$_M_DATACHECK (a third bit is reserved). IO\$_M_INHRETRY is used to inhibit all error recovery. If any error occurs, and this modifier bit is specified, the operation is terminated immediately and a failure status is returned in the I/O status block (see Section 7.9). IO\$_M_DATACHECK is used to compare the data in memory with that on a disk or magnetic tape.

7.5

Assigning Channels

Before any input or output operation can be performed on a physical device, a channel must be assigned to the device to provide a path between the process and the device. The Assign I/O Channel (\$ASSIGN) system service establishes this path.

When you write a call to the \$ASSIGN service, you must supply the name of the device, which may be a physical device name or a logical name, and the address of a word to receive the channel number. The service returns a channel number, and you use this channel number when you write an input or an output request.

For example, the following lines assign an I/O channel to the device TTA2. The channel number is returned in the word at TTCHAN.

```
TTNAME: .ASCID /TTA2:/           ; terminal descriptor
TTCHAN: .BLKW 1                   ; terminal channel number
```

```
$ASSIGN_S -
    DEVNAM=TTNAME, -
    CHAN=TTCHAN
```

To assign a channel to the current default input or output device, use the logical name SYS\$INPUT or SYS\$OUTPUT.

For more details on how \$ASSIGN and other I/O services handle logical names, see Section 7.15, Logical Names and Physical Device Names.

7.6

Queuing I/O Requests

All input and output operations in VAX/VMS are initiated with the Queue I/O Request (\$QIO) system service. The \$QIO services queues the request and returns. While the operating system processes the request, the program that issued the request can continue execution.

Required arguments to the \$QIO service include the channel number assigned to the device on which the I/O is to be performed, and a function code (expressed symbolically) that indicates the specific operation to be performed. Depending on the function code, one through six additional parameters may be required.

For example, the IO\$_WRITEVBLK and IO\$_READVBLK function codes are device-independent codes used to read and write single records or virtual blocks. These function codes are suitable for simple terminal I/O. They require parameters indicating the address of an input or output buffer and the buffer length. A call to \$QIO to write a line to a terminal might appear as follows:

```
$QIO_S  CHAN=TTCHAN, -
        FUNC=IO$_WRITEVBLK, -
        P1=BUFADDR, -
        P2=#BUFLN
```

Function codes are defined for all supported device types, and most of the codes are device dependent, that is, they perform functions that are specific to a particular device. The \$IODEF macro defines symbolic names for these function codes. See Section 2.3 for information on how to obtain a listing of these symbolic names. For details on all function codes and an explanation of the parameters required by each, see the *VAX/VMS I/O Reference Volume*.

Input/Output Services

Synchronizing Service Completion

7.7

Synchronizing Service Completion

The \$ENQ and \$QIO system services return control to the calling program as soon as a request is queued; the status code returned in R0 indicates whether or not the request was queued successfully. To ensure proper synchronization of the queuing operation with respect to the program, the program must do the following:

- 1 Test that the operation was successfully queued
- 2 Test whether the operation itself completed successfully

Optional arguments to the \$ENQ and \$QIO services provide techniques for synchronizing I/O completion. There are three methods you can use to test for the completion of a lock management or I/O request.

- Specify the number of an event flag to be set when the operation completes.
- Specify the address of an AST routine to be executed when the operation completes.
- Specify the address of an I/O status block, or lock status block, in which the system can place the return status when the operation completes.

I/O status blocks are explained in Section 7.8; lock status blocks are explained in Section 12.3.3.

Examples of using these three techniques are shown in the three examples that follow.

Example 1: Event Flags ①

```
$QIO_S EFN=#1,... ② ; issue 1st I/O request
BSBW ERROR ; queued successfully?
$QIO_S EFN=#2,... ; issue 2nd I/O request
BSBW ERROR ; queued successfully?
$WFLAND_S - ③ ; wait till both done
EFN=#0, - ③
MASK=#B110
```

- ① When you specify an event flag number as an argument, \$QIO clears the event flag when it queues the I/O request. When the I/O completes, the flag is set.
- ② In this example, the program issues two Queue I/O requests. A different event flag is specified for each request.
- ③ The Wait for Logical AND of Event Flags (\$WFLAND) system service places the process in a wait state until both I/O operations are complete. The **efn** argument indicates that the event flags are both in cluster 0; the **mask** argument indicates the flags that are to be waited for.
- ④ Note that the \$WFLAND system service (and the other wait system services) wait for the event flag to be set; they do not wait for the I/O operation to complete. If some other event were to set the required event flags, the wait for event flag would complete prematurely. Use of event flags must be carefully coordinated. (See Section 3.1 for a discussion of the recommended technique for testing I/O completion.)

Input/Output Services

Synchronizing Service Completion

Example 2: An AST Routine ①

```

$QIO_S ....,ASTADR=TTAST, - ②; I/O with AST
          ASTPRM=#1,...
BSBW      ERROR          ; queued successfully?
          ; continue

.ENTRY     TTAST,~M<R10,R11> ③; AST service routine entry mask
          ; handle I/O completion

RET                          ; end of service routine

```

- ① When you specify the **astadr** argument to the \$QIO system service, the system interrupts the process when the I/O completes and passes control to the specified AST service routine.
- ② The \$QIO system service call specifies the address of the AST routine, TTAST, and a parameter to pass as an argument to the AST service routine. When \$QIO returns control, the process continues execution.
- ③ When the I/O completes, the AST routine TTAST is called, and it responds to the I/O completion. By examining the AST parameter, TTAST can determine the origin of the I/O request.

When this routine is finished executing, control returns to the process at the point at which it was interrupted. If you specify the **astadr** argument in your call to \$QIO, you should also specify the **iosb** argument, so that the AST routine can evaluate whether or not the I/O completed successfully.

Example 3: The I/O Status Block ①

```

TTIOSB: .BLKQ 1 ② ; I/O status block

③ $QIO_S ....,IOSB=TTIOSB,... ; issue I/O request
BSBW      ERROR          ; queued successfully?
          ; continue

10$:      TSTW  TTIOSB ④ ; is I/O done yet?
          BEQL  10$      ; no, loop til done
          ⑤ CMPW  TTIOSB,#SS$ _NORMAL ; I/O successful?
          BNEQ  IO_ERR    ; no, handle the error

```

- ① An I/O status block is a quadword structure that the system uses to post the status of an I/O operation. The quadword area must be defined in your program.
- ② TTIOSB defines the I/O status block for this I/O operation. The **iosb** argument in the \$QIO system service refers to this quadword.
- ③ \$QIO clears the quadword when it queues the I/O request. When the request is queued, the program calls a routine to check whether the request was successfully placed on the queue; if queuing was successful, the program continues execution.
- ④ The process polls the I/O status block. If the low-order word still contains 0, the I/O operation has not yet completed. In this example, the program loops until the request is complete.

Input/Output Services

Synchronizing Service Completion

- 5 Once the I/O operation is complete, the process compares the low word of the I/O status block with the success status `SS$_NORMAL`. If the return status is not `SS$_NORMAL`, the program branches to `IO_ERR`.

Note: The technique shown in Example 3 wastes system time looping until the request is complete; this technique should be used only when it is the last possible alternative.

7.7.1 Recommended Method for Testing Asynchronous Completion

DIGITAL recommends that you use the Synchronize (`$SYNCH`) system service to wait for completion of an asynchronous event. The `$SYNCH` service correctly waits for the actual completion of an asynchronous event, even if some other event sets the event flag.

To use the `$SYNCH` service to wait for the completion of an asynchronous event, you must specify both an event flag number and the address of an I/O status block (IOSB) in your call to the asynchronous system service. The asynchronous service queues the request and returns control to your program. When the asynchronous service completes, it sets the event flag and places the final status of the request in the IOSB.

In your call to `$SYNCH`, you must specify the same `efn` and I/O status block that you specified in your call to the asynchronous service. `$SYNCH` waits for the event flag to be set by use of the `$WAITFR` system service. When the specified event flag is set, `$SYNCH` checks the specified I/O status block. If the I/O status block is nonzero, the system service has completed and `$SYNCH` returns control to your program. If the I/O status block is zero, `$SYNCH` clears the event flag by use of the `$CLREF` service and calls the `$WFLOP` service to wait for the event flag to be set.

`$SYNCH` sets the event flag before returning control to your program. This insures that the call to `$SYNCH` will not interfere with testing for completion of another asynchronous event that completes at approximately the same time and uses the same event flag to signal completion.

The following call to the Queue I/O Request (`$QIO`) system service demonstrates how the `$SYNCH` service is used.

```
EVENT_FLAG = 1
;
Q_IOSB: .QUAD 0
;
;
$QIO_S EFN=EVENT_FLAG, -      ; request I/O
        IOSB=Q_IOSB,...
$SYNCH_S -
        EFN=EVENT_FLAG
        IOSB=Q_IOSB          ;wait until I/O completes
BLBC    RO,ERROR             ; test status
;
;
```

Note: The `$QIOW` service provides a combination of `$QIO` and `$SYNCH`. The program segment above simply provides an example of how `$SYNCH` operates. For a more complete example see Section 2.5.

Input/Output Services

Synchronous Forms of Input/Output Services

7.8

Synchronous Forms of Input/Output Services

Some input/output services can be executed either synchronously or asynchronously. The "W" at the end of the system service name indicates the synchronous version of the system service.

The synchronous version of a system service combines the functions of the asynchronous version of the service and the Synchronize (\$SYNCH) system service. The synchronous version acts exactly as if you had used the asynchronous version of the system service followed immediately by a call to \$SYNCH; it queues the I/O request, and then places the program in a wait state until the I/O request completes. The synchronous version takes the same arguments as the asynchronous version.

The following list gives the asynchronous and synchronous names of input/output services that have synchronous versions.

Asynchronous Name	Synchronous Name	Description
\$BRKTHRU	\$BRKTHRUW	Breakthrough
\$GETDVI	\$GETDVIW	Get Device/Volume Information
\$GETJPI	\$GETJPIW	Get Job/Process Information
\$GETLKI	\$GETLKIW	Get Lock Information
\$GETQUI	\$GETQUIW	Get Queue Information
\$QIO	\$QIOW	Queue I/O Request
\$SNDJBC	\$SNDJBCW	Send to Job Controller
\$UPDSEC	\$UPDSECW	Update Section File on Disk

7.9

I/O Completion Status

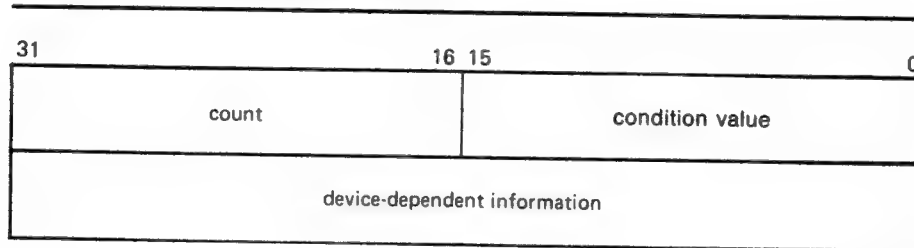
When an I/O operation completes, the system posts the completion status in the I/O status block, if one is specified. The completion status indicates whether the operation actually completed successfully, the number of bytes that were transferred, and additional device-dependent return information.

The following figure illustrates the format of the information written in the IOSB.

Input/Output Services

I/O Completion Status

Figure 7-9 I/O Status Block



ZK-856-82

The first word contains a system status code indicating the success or failure of the operation. The status codes used are the same as for all returns from system services; for example, `SS$_NORMAL` indicates successful completion.

The second word contains the number of bytes actually transferred in the I/O operation. Note that for some devices this word contains only the low-order word of the count. For information on specific devices, see the *VAX/VMS I/O Reference Volume*.

The second longword contains device-dependent return information.

To ensure successful I/O completion and the integrity of data transfers, the IOSB should be checked following I/O requests, particularly for device-dependent I/O functions. For complete details on how to use the I/O status block, see the *VAX/VMS I/O Reference Volume*.

7.10 Deassigning I/O Channels

When a process no longer needs access to an I/O device, it should release the channel assigned to the device by calling the Deassign I/O Channel (`$DASSGN`) system service.

`$DASSGN_8 CHAN=TTCHAN`

This service call releases the terminal channel assignment acquired in the `$ASSIGN` example shown earlier. The system automatically deassigns channels for a process when the image that assigned the channel exits.

7.11 Complete Terminal I/O Example

The following example shows a complete sequence of input and output operations using the `$QIOW` macro to read and write lines to the current default `SY$INPUT` device. This program will function correctly only if it is executed interactively, because the input/output must be to the current terminal.

Input/Output Services

Complete Terminal I/O Example

```

TTNAME: .ASCID /SYS$INPUT:/ ① ; descriptor for terminal name
TTCHAN: .BLKW 1 ; receive channel number here
TTIOSB: .BLKW 1 ② ; first word of IOSB, status
TTIOLEN: .BLKW 1 ; second word, get length
        .BLKL 1 ; second longword of IOSB
OUTLEN: .BLKL 1 ③ ; length of string to output
INBUF: .BLKB 80 ; buffer to read input

10$: ④$ASSIGN_S - ; assign channel
        DEVNAM=TTNAME, - ; logical name translated by $ASSIGN
        CHAN=TTCHAN
        BSBW ERROR
⑤ $QIOW_S -
        FUNC=#IO$_READVBLK-
        CHAN=TTCHAN, -
        P2=LENGTH, -
        P1=BUFFER, -
        IOSB=TTIOSB
        BSBW ERROR
        MOVZWL TTIOSB,RO ; move status code to RO
⑥ BSBW ERROR
⑦ MOVZWL TTIOLEN,OUTLEN ; get length out of IOSB
        $QIOW_S -
        FUNC=#IO$_WRITVBLK-
        CHAN=TTCHAN, -
        P2=LENGTH, -
        P1=BUFFER, -
        IOSB=TTIOSB
⑧ BSBW ERROR
        MOVZWL TTIOSB,RO ; move status code to RO
        BSBW ERROR
⑨ $DASSGN_S - ; done, deassign channel
        CHAN=TTCHAN
        BSBW ERROR

ERROR: BLBS RO,10$ ; check for successful
        ; return code
        ;
        $EXIT_S RO ; if not successful,
        ; exit and signal
        ;
10$: RSB ; if successful,
        ; return to caller

```

- ① TTNAME is a character string descriptor for the logical device SYS\$INPUT and TTCHAN is a word to receive the channel number assigned to it.
- ② The IOSB for the I/O operations is structured so that the program can easily check for the completion status (in the first word) and the length of the input string returned (in the second word).
- ③ The string will be read into the buffer INBUF; the longword OUTLEN will contain the length of the string for the output operation.
- ④ \$ASSIGN assigns a channel and writes the channel number at TTCHAN.
- ⑤ If the \$ASSIGN service completes successfully, the \$QIOW macro reads a line from the terminal, and requests that the completion status be posted in the I/O status block defined at TTIOSB.

Input/Output Services

Complete Terminal I/O Example

- ⑥ The process waits until the I/O is complete, then checks the first word in the I/O status block for a successful return. If not successful, the program takes an error path.
- ⑦ Next, the length of the string read is moved into the longword at OUTLEN. This is necessary because the \$QIOW macro requires a longword argument, but the length field of the I/O status block is only a word long. The \$QIOW macro writes the line just read to the terminal.
- ⑧ The program performs error checks: first, it ensures that the \$OUTPUT macro successfully queued the I/O request; then, when the request is completed, it ensures that the I/O was successful.
- ⑨ When all I/O operations on the channel are finished, the channel is deassigned.

7.12 Canceling I/O Requests

If a process must cancel I/O requests that have been queued but not yet completed, it can issue the Cancel I/O On Channel (\$CANCEL) system service. All pending I/O requests issued by the process on that channel are canceled; you cannot specify a particular I/O request.

For example, the \$CANCEL system service can be called as follows:

```
$CANCEL_S CHAN=TTCHAN
```

In this example, the \$CANCEL system service initiates the cancellation of all pending I/O requests to the channel whose number is located at TTCHAN.

The \$CANCEL system service returns after initiating the cancellation of the I/O requests. If the call to \$QIO specified an event flag, AST service routine, or I/O status block, the system sets the flag, delivers the AST, or posts the I/O status block as appropriate when the cancellation is actually completed.

7.13 Device Allocation

Many I/O devices are shareable; that is, more than one process can access the device at a time. By calling the Assign I/O Channel (\$ASSIGN) system service, a process is given a channel to the device for I/O operations.

In some cases, a process may need exclusive use of a device so that data is not affected by other processes. To reserve a device for exclusive use you must allocate it.

Device allocation is normally accomplished from the DCL command stream, with the ALLOCATE command. A process can also allocate a device by calling the Allocate Device (\$ALLOC) system service. When a device has been allocated by a process, only the process that allocated the device and any subprocesses it creates can assign channels to the device.

When you call the \$ALLOC system service, you must provide a device name. The device name specified can be any of the following:

- A physical device name, for example, the tape drive MTB3:
- A logical name, for example, TAPE
- A generic device name, for example, MT:

Input/Output Services

Device Allocation

If you specify a physical device name, \$ALLOC attempts to allocate the specified device.

If you specify a logical name, \$ALLOC translates the logical name and attempts to allocate the physical device name equated to the logical name.

If you specify a generic device name, that is, if you specify a device type but do not specify a controller and/or unit number, \$ALLOC attempts to allocate any device available of the specified type. More information on the allocation of devices by generic names is provided in Section 7.15.1.

When you specify generic device names, you must provide fields for the \$ALLOC system service to return the name and the length of the physical device that is actually allocated, so that you can provide this name as input to the \$ASSIGN system service.

The following example illustrates the allocation of a tape device specified by the logical name TAPE.

```
LOGDEV: .ASCID /TAPE/           ; descriptor for logical name
DEVDESC:                          ; descriptor for physical name
      .LONG 64                  ; length of buffer
      .ADDRESS -                ; address of buffer
      DEVSTR
DEVSTR: .BLKB 64                ; get physical name returned
TAPECHAN:
      .BLKW 1                   ; channel for tape I/O
```

```
① $ALLOC_S -
      DEVNAM=LOGDEV, -
      PHYLEN=DEVDESC, -
      PHYBUF=DEVDESC
      BSBW ERROR
② $ASSIGN_S -                    ; assign channel
      DEVNAM=DEVDESC, -
      CHAN=TAPECHAN
      BSBW ERROR
      ; continue with I/O
③ $DASSGN_S -                   ; deassign channel
      CHAN=TAPECHAN
      BSBW ERROR
      $DALLOC_S -               ; deallocate tape
      DEVNAM=DEVDESC
```

- ① The \$ALLOC system service call requests allocation of a device corresponding to the logical name TAPE, defined by the character string descriptor LOGDEV. The argument DEVDESC refers to the buffer provided to receive the physical device name of the device actually allocated and the length of the name string. \$ALLOC translates the logical name TAPE, and returns the equivalence name string of the device actually allocated into the buffer at DEVDESC. It writes the length of the string in the first word of DEVDESC.
- ② The \$ASSIGN command uses the character string returned by the \$ALLOC system service as the input device name argument, and requests that the channel number be written into TAPECHAN.
- ③ When I/O operations are completed, the \$DASSGN system service deassigns the channel and the \$DALLOC system service deallocates the device. The channel must be deassigned before the device can be deallocated.

Input/Output Services

Device Allocation

7.13.1 Implicit Allocation

Devices that cannot be shared by more than one process (for example, terminals and line printers) do not have to be explicitly allocated. Since they are nonshareable, they are implicitly allocated by the \$ASSIGN system service when \$ASSIGN is called to assign a channel to the device.

7.13.2 Deallocation

When the program has finished using an allocated device, it should release the device with the Deallocate Device (\$DALLOC) system service, to make it available for other processes as in this example:

```
$DALLOC_S DEVNAM=DEVDESC
```

At image exit, the system automatically deallocates devices allocated by the image.

7.14 Mounting and Dismounting Volumes

Mounting a volume establishes a link between a volume, a device, and a process. A volume, or volume set, must be mounted before I/O operations can be performed on the volume. Normally, a user mounts or dismounts a volume from the DCL command stream with the MOUNT or DISMOUNT commands. A process can also mount a volume or volume set using the Mount Volume (\$MOUNT) system service. A volume or volume set can be dismounted with the Dismount Volume (\$DISMOU) system service.

Mounting a volume involves two distinct operations.

- Placing the volume on the device and starting the device (by pressing the START or LOAD button).
- Mounting the volume with the \$MOUNT system service.

7.14.1 Calling the \$MOUNT System Service

The Mount Volume (\$MOUNT) system service allows a process to mount a single volume, or a volume set. When you call the \$MOUNT system service, you must specify a device name.

The \$MOUNT system service has a single argument, an address of a list of item descriptors. The list is terminated by a longword of binary zeros. Figure 7-10 shows the format of an item descriptor.

Input/Output Services

Mounting and Dismounting Volumes

Figure 7-10 \$MOUNT Item Descriptor

31	15	0
item code	buffer length	
buffer address		
return length address		

ZK-1705-84

Most item descriptors do not have to be in any order. To mount volume sets, you must specify one item descriptor per device and one item descriptor per volume; the descriptors for the volumes must be specified in the same order as the descriptors for the devices on which the volumes are loaded.

For item descriptors other than device and volume name, if you specify the same item descriptor more than once, the last occurrence of the descriptor is used.

The following example illustrates a call to \$MOUNT. The call is equivalent to the DCL command:

```
$ MOUNT/SYSTEM/NOQUOTA DRA4:,DRA5: USER01,USER02 USERD$
$MNTDEF
ITEMS: .WORD 4 ; length of flags
        .WORD MNT$_FLAGS ; flag code
        .ADDRESS - ; address of flags longword
        FLAGS
        .LONG 0 ; unused longword
;
        .WORD 5 ; length of first device name
        .WORD MNT$_DEVNAM ; device code
        .ADDRESS - ; address of first device name
        DEV1
        .LONG 0 ; unused longword
;
        .WORD 6 ; length of first volume name
        .WORD MNT$_VOLNAM ; volume code
        .ADDRESS - ; address of first volume name
        VOL1
        .LONG 0 ; unused longword
;
        .WORD 5 ; length of second device name
        .WORD MNT$_DEVNAM ; device code
        .ADDRESS - ; address of second device name
        DEV2
        .LONG 0 ; unused longword
;
        .WORD 6 ; length of second volume name
        .WORD MNT$_VOLNAM ; volume code
        .ADDRESS - ; address of second volume name
        VOL2
        .LONG 0 ; unused longword
;
        .WORD 6 ; length of volume logical name
        .WORD MNT$_LOGNAM ; logical name code
        .ADDRESS - ; address of volume logical name
        LOG
        .LONG 0 ; unused longword
        .LONG 0 ; end of item list
;
```

Input/Output Services

Mounting and Dismounting Volumes

```
DEV1: .ASCII /DRA4:/          ; first device
VOL1: .ASCII /USER01/         ; first volume name
DEV2: .ASCII /DRA5:/          ; second device
VOL2: .ASCII /USER02/         ; second volume name
LOG: .ASCII /USERD$/          ; logical name
;
FLAGS: .LONG <MNT$M_SYSTEM!MNT$M_NODISKQ>
;
;
$MOUNT_S -                    ; now call $MOUNT
ITMLST=ITEMS
;
;
;
```

7.14.2 Calling the \$DISMOU System Service

The \$DISMOU system service allows a process to dismount a volume or volume set. When you call \$DISMOU, you must specify a device name. If the volume mounted on the device is part of a fully mounted volume set, and no flags are specified, the whole volume set is dismounted.

The following example illustrates a call to \$DISMOU. The call dismounts the volume set mounted in the previous example.

```
DEV1_DESC:
.ASCID /DRA4:/
;
;
$DISMOU_S -
DEVNAM=DEV1_DESC
;
;
;
```

7.15 Logical Names and Physical Device Names

When a device name is specified as input to an I/O system service, it can be a physical device name or a logical name. If the device name contains a colon, the colon and the characters past it are ignored. When an underscore character (—) precedes a device name string, it indicates that the string is a physical device name string. For example:

```
TTNAME: .ASCID /_TTB3:/
```

Any string that does not begin with an underscore is considered a logical name, even though it may be a physical device name. The following system services translate a logical name iteratively until a physical device name is returned, or until the system default number of translations have been performed.

- Allocate Device (\$ALLOC)
- Assign I/O Channel (\$ASSIGN)
- Broadcast (\$BRDCST)
- Deallocate Device (\$DALLOC)
- Dismount Volume (\$DISMOU)
- Get I/O Device Information (\$GETDEV)

Input/Output Services

Logical Names and Physical Device Names

- Get Device/Volume Information (\$GETDVI)
- Mount Volume (\$MOUNT)

In each translation, the logical name tables defined by the logical name LNM\$FILE_DEV are searched in order. These tables, listed in search order, are normally LNM\$PROCESS, LNM\$JOB, LNM\$GROUP and LNM\$SYSTEM. If a physical device name is located, the I/O request is performed for that device.

If the services do not locate an entry for the logical name, the I/O service treats the name that is specified as a physical device name. When you specify the name of an actual physical device in a call to one of these services, include the underscore character to bypass the logical name translation.

When the \$ALLOC system service returns the device name of the physical device that has been allocated, the device name string returned is prefaced with an underscore character. When this name is used for the subsequent \$ASSIGN system service, the \$ASSIGN service does not attempt to translate the device name.

If you use logical names in I/O service calls, you must be sure to establish a valid device name equivalence before program execution. You can do this by issuing a DEFINE command from the command stream, or by having the program establish the equivalence name before the I/O service call with the Create Logical Name (\$CRELNM) system service.

For details on how to create and use logical names, see Section 6, Logical Name Services.

7.15.1 Device Name Defaults

If, after logical name translation, a device name string in an I/O system service call does not fully specify the device name (that is, device, controller, and unit), the service either provides default values for nonspecified fields, or provides values based on device availability.

The following rules apply:

- The \$ASSIGN and \$DALLOC system services apply default values as shown in Table 7-2.
- The \$ALLOC system service treats the device name as a generic device name and attempts to find a device that satisfies the components of the device name that is specified, as shown in Table 7-2.

7.16 Obtaining Information About Physical Devices

The Get Device/Volume Information (\$GETDVI) returns information about devices. The information returned is specified by an item list created before the call to \$GETDVI.

When you call the \$GETDVI system service, you must provide the address of an item list that specifies the information to be returned. The format of the item list is described in the description of \$GETDVI given in Part II. Details on the device-specific information these services return is given in the *VAX/VMS I/O Reference Volume*.

Input/Output Services

Obtaining Information About Physical Devices

In cases where a generic (that is, nonspecific) device name is used in an I/O service, a program might need to find out what device has actually been used. To do this, the program should provide \$GETDVI with the number of the channel to the device and request the name of the device with the DVI\$_DEVNAM item identifier.

Table 7-2 Default Device Names for I/O Services

Device Name Specified	Device Name Defaults for Most I/O Services ¹	Generic Device Names Used by \$ALLOC and \$MOUNT
dd:	ddA0: (unit 0 on controller A)	ddxy: (any available device of the specified type)
ddc:	ddc0: (unit 0 on controller specified)	ddcy: (any available unit on the specified controller)
ddu:	ddAu: (unit specified on controller A)	ddxu: (device of specified type and unit on any available controller)
ddcu:	ddcu: (unit and controller specified)	ddcu: (unit and controller specified)

¹Capital letters indicate a specific controller; numbers indicate a specific unit.

Key:

dd: is the device type specified ¹
c: is the controller specified
x: is any controller
u: is the unit number specified
y: is any unit number

¹A summary of the device names is contained in the *VAX/VMS DCL Concepts Manual*.

VAX/VMS also supports a device for program development called the null device. NL is the mnemonic for the null device. Its characteristics are as follows:

- A read from NL returns an end-of-file error (SS\$_ENDOFFILE).
- A write to NL immediately returns a success indication (SS\$_NORMAL).

The null device functions as a virtual device to which you can direct output, but from which the data does not return.

7.17 Formatting Output Strings

When you are preparing output strings for a program, you may need to insert variable information into a string prior to output, or you may need to convert a numeric value to an ASCII string. The Formatted ASCII Output (\$FAO) system service performs these functions.

Input to the \$FAO service consists of the following:

- A control string that contains the fixed text portion of the output and formatting directives. The directives indicate the position within the string where substitutions are to be made, and describe the data type and length of the input values that are to be substituted or converted.
- An output buffer to contain the string after conversions and substitutions have been made.
- An optional argument indicating a word to receive the final length of the formatted output string.
- Parameters that provide arguments for the formatting directives.

The following example shows a call to the \$FAO system service to format an output string for a \$QIOW macro. Accompanying notes briefly discuss the input and output requirements of \$FAO. Complete details on how to use \$FAO, with additional examples, are provided in the description of the \$FAO system service in Part II.

```

FAOSTR: ① .ASCID /FILE !AS DOES NOT EXIST/ ; descriptor for
; FAO control string
FAODESC: ② ; descriptor for $FAO output
.LONG 80 ; length of buffer
.ADDRESS - ; address of buffer
FAOBUF
FAOBUF: .BLKB 80 ; buffer for $FAO output
FAOLEN: .LONG 0 ; receive length of $FAO output
FILESPEC: ③
.ASCID /DISK$USER:MYFILE.DAT/ ; descriptor for FAO parameter

```

```

④ $FAO_S CTRSTR=FAOSTR, -
        OUTLEN=FAOLEN, -
        OUTBUF=FAODESC, -
        PI=FILESPEC ; parameter for $FAO
        BSBW ERROR
⑤ $QIOW ...,BUFFER=FAOBUF, -
        LENGTH=FAOLEN
        BSBW ERROR

```

- 1 FAOSTR provides the FAO control string. !AS is an example of an FAO directive: it requires an input parameter that specifies the address of a character string descriptor. When \$FAO is called to format this control string, !AS will be substituted with the string whose descriptor address is specified.
- 2 FAODESC is a character string descriptor for the output buffer; \$FAO will write the string into the buffer, and will write the length of the final formatted string in the low-order word of FAOLEN. (A longword is reserved so that it can be used for an input argument to the \$QIOW macro.)
- 3 FILESPEC is a character string descriptor defining an input string for the FAO directive !AS.

Input/Output Services

Formatting Output Strings

- ④ The call to \$FAO specifies the control string, the output buffer and length fields, and the parameter P1, which is the address of the string descriptor for the string to be substituted.
- ⑤ When \$FAO completes successfully, \$QIOW writes the following output string:

FILE DISK\$USER:MYFILE.DAT DOES NOT EXIST

7.18 Mailboxes

Mailboxes are virtual devices that can be used for communication among processes. Actual data transfer is accomplished by using VAX RMS or I/O services. When the create Mailbox and Assign Channel (\$CREMBX) service creates a mailbox, it also assigns a channel to it for use by the creating process. Other processes can then assign channels to the mailbox using either the \$CREMBX or \$ASSIGN system service. (If the mailbox is located in memory shared by multiple processors, the first process on each processor to create or assign a channel to a mailbox must use the \$CREMBX service.)

The \$CREMBX system service creates the mailbox. The \$CREMBX system service identifies a mailbox by a user-specified logical name and assigns it an equivalence name. The equivalence name is a physical device name in the format MBAn: where n is a unit number. The equivalence name has the terminal attribute.

When another process assigns a channel to the mailbox with the \$CREMBX or \$ASSIGN system service, it can identify the mailbox by its logical name. The service automatically translates the logical name. The process can obtain the MBAn: name by translating the logical name (with the \$TRNLNM system service), or it can call the Get Device/Volume Information (\$GETDVI) system service to obtain the unit number and the physical device name.

Mailboxes are either temporary or permanent. The user privileges TMPMBX and PRMMBX are required to create temporary and permanent mailboxes, respectively.

For a temporary mailbox, the \$CREMBX service enters the logical name and equivalence name in the logical name table LNM\$TEMPORARY_MAILBOX. This logical name table name usually specifies the LNM\$JOB logical name table name. The system deletes a temporary mailbox when no more channels are assigned to it.

For a permanent mailbox, the \$CREMBX service enters the logical name and equivalence name in the logical name table LNM\$PERMANENT_MAILBOX. This logical name table name usually specifies the LNM\$SYSTEM logical name table name. Permanent mailboxes continue to exist until they are specifically marked for deletion with the Delete Mailbox (\$DELMBX) system service.

A mailbox located in memory shared by multiple processors is also deleted when all of the following occur:

- A processor is rebooted
- The multiport memory is not reinitialized
- No other processor has any processes with channels assigned to the mailbox

Input/Output Services

Mailboxes

The following example shows how processes can communicate by means of a mailbox. The accompanying notes explain some of the arguments that the \$CREMBX system service requires.

```

Process ORION
MBLOGNAM:      ; mailbox logical
               .ASCID /GROUP100__MAILBOX/ ; name descriptor
MBUFLEN = 128
MBUFFER:
               .BLKB MBUFLEN ; input buffer for mailbox reads
MBXCHAN:
               .BLKW 1 ; mailbox channel number
MBXIOSB:
               .BLKW 1 ; IO SB first word (status)
MBLEN: .BLKW 1 ; IO SB 2nd word (length)
               .BLKL 1 ; remainder of IO SB
OUTLEN: .BLKL 1 ; longword to get length

               .ENTRY ORION,_"M<R2,R3,R4>" ; entry mask
;
1  $CREMBX__S -
               PRMFLG=_#0, -
               CHAN=MBXCHAN, -
               MAXMSG=#MBUFLEN, -
               BUFQUO=#384, -
               PROMSK=_#_X0000, -
               LOGNAM=MBLOGNAM
               BSBW ERROR
;
2  $QIO__S CHAN=MBXCHAN, -
               FUNC=_#IO$__READVBLK, -
               IO SB=MBXIOSB, -
               ASTADR=MBXAST, -
               P1=MBUFFER, -
               P2=#MBUFLEN
               BSBW ERROR
;
               RET
;
3  .ENTRY MBXAST,_"M<R2,R3,R4>" ; AST routine entry mask
;
               CMPW MBXIOSB,_#SS$__NORMAL ; I/O successful?
               BNEQ ASTERR ; branch if not
               MOVZWL MBLEN,OUTLEN ; make length a longword
               $QIO__S ...,BUFFER=MBUFFER, -
               LENGTH=OUTLEN,...
               BSBW ERROR
;
               RET
Process CYGNUS
MAILBOX:      ; mailbox logical name descriptor
               .ASCID /GROUP100__MAILBOX/
MAILCHAN:      ; mailbox channel number
               .BLKW 1
OUTBUF: .BLKB 128 ; buffer for output msg data
OUTLEN: .BLKL 1 ; will contain length of msg
;
               .ENTRY CYGNUS,_"M<R2,R3,R4>" ; entry mask

```

Input/Output Services

Mailboxes

```
❶ $ASSIGN__S - ; assign channel
    DEVNAM=MAILBOX, -
    CHAN=MAILCHAN
    BSBW ERROR
.
.
$QIOW_S CHAN=MAILCHAN, -
    BUFFER=OUTBUF, -
    LENGTH=OUTLEN,...
    BSBW ERROR
.
RET
```

- ❶ Process ORION creates the mailbox and receives the channel number at MBXCHAN.

This **prmlg** argument indicates that the mailbox is a temporary mailbox. The logical name is entered in the LNM\$TEMPORARY_MAILBOX logical name table.

The **maxmsg** argument limits the size of messages that the mailbox can receive. Note that the size indicated in this example is the same size as the buffer (MBUFFER) provided for the \$QIO request. A buffer for mailbox I/O must be at least as large as the size specified in the MAXMSG argument.

When a process creates a temporary mailbox, the amount of system memory that is allocated for buffering messages is subtracted from the process's buffer quota. Use the BUFQUO argument to specify how much of the process quota you want to be used for mailbox message buffering.

Mailboxes are protected devices. By specifying a protection mask with the **promsk** argument, you can restrict access to the mailbox. (In this example, all bits in the mask are clear, indicating unlimited read and write access.)

- ❷ After creating the mailbox, process ORION calls the \$QIO system service, requesting that it be notified when I/O completes (that is, when the mailbox receives a message) by means of an AST interrupt. The process can continue executing, but the AST service routine at MBXAST will interrupt and begin executing when a message has been received.
- ❸ When a message is sent to the mailbox (by CYGNUS), the AST is delivered and ORION responds to the message. ORION gets the length of the message from the first word of the I/O status block at MBXIOSB and places it in the longword OUTLEN so it can pass the length to \$QIOW_S.
- ❹ Process CYGNUS assigns a channel to the mailbox, specifying the logical name the process ORION gave the mailbox. The \$QIOW system service writes a message from the output buffer provided at OUTBUF.

Note that on a write operation to a mailbox, the I/O is not complete until the message is read, unless you specify the IO\$M_NOW function modifier. Therefore, if \$QIOW (without the IO\$M_NOW function modifier) is used to write the message, the process will not continue executing until another process reads the message.

7.18.1 Mailbox Name Format

The logical name string assigned to a mailbox determines whether it is located in memory that is used by a single processor or in memory that is shared by multiple processors. The **lognam** argument to the \$CREMBX service specifies a descriptor that points to a character string with the following format:

`[shared-memory-name:]mailbox-name`

- The shared-memory-name locates the mailbox within the named memory that is shared by multiple processors. The name of this memory was specified at system generation time. For example, the string SHRMEM\$1:CHKPNT identifies a mailbox named CHKPNT located in the shared memory named SHRMEM\$1.

If this part of the string is not included, the mailbox is not located in memory that is shared by multiple processors.

- The mailbox-name is the name assigned to the mailbox (1 to 15 characters in length).

If you wish, you can include both the shared-memory-name and the mailbox-name for a mailbox in memory shared by multiple processors. However, if you want to use existing programs without recompiling or relinking, you can specify just a mailbox-name and have the system translate it to a complete specification. The system attempts to perform logical name translation of the string specified by the **lognam** argument in the following manner.

- The current name string is searched for a colon. If a colon is found within the current name string, the mailbox is assumed to be located in shared memory and translation proceeds in the following manner.
 - 1 The portion of the current name string to the right of the colon is placed in the mailbox-name buffer. The portion of the current name string to the left of the colon becomes the new current name string.
 - 2 MBX\$ is prefixed to the current name string and the result is subjected to logical name translation.
 - 3 If the result contains a logical name, steps 1 and 2 are repeated until translation does not succeed or until the number of translations performed exceeds the number specified by the SYSGEN parameter LNM\$_MAXDEPTH.
 - 4 The MBX\$ prefix is stripped from the current name string that could not be translated. This name becomes the shared-memory name. The current string contained in the mailbox-name buffer is made a logical name with an equivalence name MBAn: (n is a number assigned by the system).
- If the mailbox is located in local memory, translation proceeds in the following manner.
 - 1 MBX\$ is prefixed to the current name string and the result is subjected to logical name translation.
 - 2 If the result is a logical name, step 1 is repeated until translation does not succeed or until the number of translations performed exceeds the number specified by the SYSGEN parameter LNM\$_MAXDEPTH.

Input/Output Services

Mailboxes

- 3 The MBX\$ prefix is stripped from the current name string that could not be translated. This current string is made a logical name with an equivalence name MBAn: (n is a number assigned by the system.)

For example, assume that you have made the following logical name assignment:

```
$ DEFINE MBX$CHKPNT SHRMEM$1:CHKPNT
```

Assume that your program also contains the following statements:

```
MBXDESC: .ASCID /CHKPNT/ ;descriptor for mailbox logical name
```

```
$CREMBX_8 LOGNAME=MBXDESC,...
```

The following logical name translation takes place:

- 1 MBX\$ is prefixed to CHKPNT.
- 2 MBX\$CHKPNT is translated to SHRMEM\$1:CHKPNT.

Since no further translation is successful, the logical name CHKPNT is created with the equivalence name MBAn: ("n" is a number assigned by the system).

There are two exceptions to the logical name translation method discussed in this section:

- If the name string starts with an underscore (_), VAX/VMS strips the underscore and considers the resultant string to be the actual name (that is, no further translation is performed).
- If the name string is the result of a logical name translation, then the name string is checked to see if it has the "terminal" attribute. If the name string is marked with the "terminal" attribute, VAX/VMS considers the resultant string to be the actual name (that is, no further translation is performed).

7.18.2 System Mailboxes

The system uses mailboxes for communication among system processes. All system mailbox messages contain, in the first word of the message, a constant that identifies the sender of the message. These constants have symbolic names (defined in the \$MSGDEF macro) in the following format:

MSG\$_sender

The following list contains the symbolic names included in the \$MSGDEF macro and describes their meanings.

Symbolic Name	Meaning
MSG\$_TRMUNSOLIC	Unsolicited terminal data
MSG\$_CRUNSOLIC	Unsolicited card reader data
MSG\$_ABORT	Network partner aborted link
MSG\$_CONFIRM	Network connect confirm
MSG\$_CONNECT	Network inbound connect initiate
MSG\$_DISCON	Network partner disconnected—hang-up

Symbolic Name	Meaning
MSG\$_EXIT	Network partner exited prematurely
MSG\$_INTMSG	Network interrupt message; unsolicited data
MSG\$_PATHLOST	Network path lost to partner
MSG\$_PROTOCOL	Network protocol error
MSG\$_REJECT	Network connect reject
MSG\$_THIRDPARTY	Network third party disconnect
MSG\$_TIMEOUT	Network connect timeout
MSG\$_NETSHUT	Network shutting down
MSG\$_NODEACC	Node has become accessible
MSG\$_NODEINACC	Node has become inaccessible
MSG\$_EVTAVL	Events are available to DECnet Event Logger
MSG\$_EVTRCVCHG	Event receiver database change
MSG\$_INCDAT	Unsolicited incoming data available
MSG\$_RESET	Request to reset the virtual circuit
MSG\$_LINUP	PVC line up
MSG\$_LINDWN	PVC line down
MSG\$_EVTXMTCHG	Event transmitter database change

The remainder of the message contains variable information, depending on the system component that is sending the message.

The format of the variable information for each message type is documented with the system function that uses the mailbox.

7.18.3 Mailboxes for Process Termination Messages

When a process creates another process, it can specify the unit number of a mailbox as an argument to the Create Process (\$CREPRC) system service. When the created process is deleted, the system sends a message to the specified termination mailbox. An example of how to create and use a termination mailbox is provided in Section 8.7.2, Termination Mailboxes.

A mailbox in memory shared by multiple processors cannot be used as a process termination mailbox.

7.18.4 Mailboxes for System Processes

Certain I/O services are used internally by system processes to communicate various kinds of information. These services are:

- Send Message to Job Controller (\$SNDJBC)
- Send Message to Operator (\$SNDOPR)

Details on the formats of the messages and the information they provide are given in the descriptions of these system services in Part II.

Input/Output Services

Example of Using I/O Services

7.19 Example of Using I/O Services

```

                                SEND.FOR
INTEGER STATUS
! Name and channel number for mailbox
CHARACTER*(*) MBX_NAME
PARAMETER (MBX_NAME = 'MAIL_BOX')
INTEGER*2 MBX_CHAN
! Mailbox message
CHARACTER*80 MBX_MESSAGE
INTEGER LEN
CHARACTER*80 MESSAGES (255)
INTEGER MESSAGE_LEN (255)
INTEGER MAX_MESSAGE
PARAMETER (MAX_MESSAGE = 255)
! I/O function codes and status block
INCLUDE '($IODEF)'
INTEGER*4 WRITE_CODE
INTEGER*2 IOSTAT,
2      MSG_LEN
INTEGER READER_PID
COMMON /IOBLOCK/ IOSTAT,
2      MSG_LEN,
2      READER_PID
! system routines
INTEGER SYS$CREMBX,
2      SYS$ASCEFC,
2      SYS$WAITFR,
2      SYS$QIOW
! Create the mailbox
STATUS = SYS$CREMBX (,
2      MBX_CHAN,
2      '...',
2      MBX_NAME)
IF (.NOT. STATUS) CALL LIB$SIGNAL (XVAL(STATUS))
! fill MESSAGES array
.
.
.
! Write the messages
DO I = 1, MAX_MESSAGE
WRITE_CODE = IO$_WRITEVBLK .OR. IO$_M_NOW
MBX_MESSAGE = MESSAGES(I)
LEN = MESSAGE_LEN(I)
STATUS = SYS$QIOW (,
2      XVAL(MBX_CHAN),      ! channel
2      XVAL(WRITE_CODE),    ! I/O code
2      IOSTAT,              ! status block
2      ..
2      XREF(MBX_MESSAGE),    ! P1
2      XVAL(LEN),...)       ! P2
IF (.NOT. STATUS) CALL LIB$SIGNAL (XVAL(STATUS))
IF (.NOT. IOSTAT) CALL LIB$SIGNAL (XVAL(STATUS))
END DO
! write end of file
WRITE_CODE = IO$_WRITEOF .OR. IO$_M_NOW
STATUS = SYS$QIOW (,
2      XVAL(MBX_CHAN),      ! channel
2      XVAL(WRITE_CODE),    ! end of file code
2      IOSTAT,              ! status block
2      ..)
IF (.NOT. STATUS) CALL LIB$SIGNAL (XVAL(STATUS))
IF (.NOT. IOSTAT) CALL LIB$SIGNAL (XVAL(STATUS))
.
.
.
! make sure cooperating process can read the information

```

Input/Output Services

Example of Using I/O Services

```

! by waiting for it to assign a channel to the mailbox
STATUS = SYS$ASCEFC (%VAL(64),
2      'CLUSTER',..)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
STATUS = SYS$WAITFR (%VAL(64))
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
END

RECEIVE.FOR

INTEGER STATUS
INCLUDE '(%IODEF)'
INCLUDE '(%SSDEF)'

! Name and channel number for mailbox
CHARACTER*(*) MBX_NAME
PARAMETER (MBX_NAME = 'MAIL_BOX')
INTEGER*2 MBX_CHAN

! QIO function code
INTEGER READ_CODE

! Mailbox message
CHARACTER*80 MBX_MESSAGE
INTEGER*4 LEN

! Message arrays
CHARACTER*80 MESSAGES (255)
INTEGER*4 MESSAGE_LEN (255)

! I/O status block
INTEGER*2 IOSTAT,
2      MSG_LEN
INTEGER READER_PID
COMMON /IOBLOCK/ IOSTAT,
2      MSG_LEN,
2      READER_PID

! system routines
INTEGER SYS$ASSIGN,
2      SYS$ASCEFC,
2      SYS$SETEF,
2      SYS$QIOW

! Create the mailbox and let the other process know
STATUS = SYS$ASSIGN (MBX_NAME,
2      MBX_CHAN,...)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
STATUS = SYS$ASCEFC (%VAL(64),
2      'CLUSTER',..)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
STATUS = SYS$SETEF (%VAL(64))
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))

! Read first message
READ_CODE = IO$_READVBLK .OR. IO$_M_NOW
LEN = 80
STATUS = SYS$QIOW (,
2      %VAL(MBX_CHAN),      ! channel
2      %VAL(READ_CODE),    ! function code
2      IOSTAT,              ! status block
2      '',
2      %REF(MBX_MESSAGE),   ! P1
2      %VAL(LEN),...)       ! P2
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
IF ((.NOT. IOSTAT) .AND.
2 (IOSTAT .NE. SS$_ENDOFFILE)) THEN
    CALL LIB$SIGNAL (%VAL(IOSTAT))
ELSE IF (IOSTAT .NE. SS$_ENDOFFILE) THEN
    I = 1
    MESSAGES(I) = MBX_MESSAGE
    MESSAGE_LEN(I) = MSG_LEN
END IF

! Read messages until cooperating process writes end-of-file
DO WHILE (.NOT. ((IOSTAT .EQ. SS$_ENDOFFILE) .AND.
2      (READER_PID .NE. 0)))

STATUS = SYS$QIOW (,

```


Input/Output Services

Example of Using I/O Services

```
2          %VAL(MBX_CHAN),      ! channel
2          %VAL(READ_CODE),     ! function code
2          IOSTAT,              ! status block
2          ..
2          %REF(MBX_MESSAGE),    ! P1
2          %VAL(LEN),...        ! P2
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
IF ((.NOT. IOSTAT) .AND.
2  (IOSTAT .NE. SS$ENDOFFILE)) THEN
  CALL LIB$SIGNAL (%VAL(IOSTAT))
ELSE IF (IOSTAT .NE. SS$ENDOFFILE) THEN
  I = I + 1
  MESSAGE(I) = MBX_MESSAGE
  MESSAGE_LEN(I) = MSG_LEN
END IF
END DO
```

In the preceding FORTRAN example, the first program creates a mailbox named MAIL_BOX, writes data to it, and then indicates the end of the data by writing an end-of-file message.

The second program creates a mailbox with the same logical name. It reads the messages from the mailbox into an array. It stops the read operations when a read operation generates an end-of-file message and the second longword of the I/O status block is non-zero. By checking that the I/O status block is non-zero, the second program confirms that the writing process sent the end-of-file message.

The processes use common event flag number 64 to ensure that SEND.FOR does not exit until RECEIVE.FOR has established a channel to the mailbox. (If RECEIVE.FOR executes first, an error occurs because SYS\$ASSIGN cannot find the mailbox.)

8

Process Control Services

When you log in to the system, the system creates a process for the execution of program images. You can create another process to execute an image by issuing the RUN or SPAWN command using any of the qualifiers that pertain to process creation. You can also write a program that creates another process to execute a particular image.

The following services are process control system services.

- Create Process (\$CREPRC)
- Delete Process (\$DELPRC)
- Suspend Process (\$SUSPND)
- Resume Process (\$RESUME)
- Hibernate (\$HIBER)
- Wake (\$WAKE)
- Schedule Wakeup (\$SCHDWK)
- Cancel Wakeup (\$CANWAK)
- Exit (\$Exit)
- Force Exit (\$FORCEX)
- Declare Exit Handler (\$DCLEXH)
- Cancel Exit Handler (\$CANEXH)
- Set Process Name (\$SETPRN)
- Set Priority (\$SETPRI)
- Set Privileges (\$SETPRV)
- Set Resource Wait Mode (\$SETRWM)
- Get Job/Process Information (\$GETJPI)

Process control services allow you to create processes and to control a process or group of processes. This section describes some aspects of process control services and includes discussions of the following:

- Subprocesses and detached processes
- The execution context of a process
- Process creation
- Interprocess control and communication
- Process hibernation and suspension
- Image exit and exit handlers
- Process deletion and termination messages

Process Control Services

Subprocesses and Detached Processes

8.1 Subprocesses and Detached Processes

A process is either a subprocess or a detached process. A subprocess receives a portion of its creator's resource quotas and must terminate before the creator. A detached process is fully independent; for example, the process the system creates at login is a detached process.

The Create Process (\$CREPRC) system service creates both subprocesses and detached processes. The number of subprocesses a process can create is controlled by its PRCLM quota. The ability to create a detached process with a UIC that is different from the UIC of the creating process is controlled by the DETACH privilege.

8.2 The Execution Context of a Process

The execution context of a process defines a process to the system. It includes the following:

- The image that the process is executing
- The input and output streams for the image executing in the process
- Disk and directory defaults for the process
- System resource quotas and user privileges available to the process

When the system creates a detached process as the result of a login, it uses the system user authorization file (SYSUAF.DAT) to determine the process's execution context.

For example, the following occurs when you log in to the system:

- The process created for you executes the image known as LOGINOUT.
- The terminal you are using is established as the input, output, and error stream device for images that the process executes.
- Your disk and directory defaults are taken from the user authorization file.
- The resource quotas and privileges you have been granted by the system manager are associated with the created process.
- A command language interpreter is mapped into the created process.

When you call the \$CREPRC system service to create a process, you define the context by specifying arguments to the service.

8.3 Process Creation

Sections 8.3.1 through 8.3.5 show examples of process creation and describe how the arguments to the \$CREPRC system service define the context of the process.

8.3.1 Defining an Image for a Subprocess to Execute

When you call the \$CREPRC system service, use the **image** argument to provide the process with the name of an image to execute. For example, the following lines create a subprocess to execute the image named CARRIE.EXE.

```
PROGNAME:
    .ASCID /CARRIE/          ; descriptor for image to execute
    .
    $CREPRC_S -              ; create process to execute CARRIE
        IMAGE=PROGNAME
```

In this example, only a file name is specified; the service uses current disk and directory defaults, performs logical name translation, uses the default file type of EXE, and locates the most recent version of the image file. When the subprocess completes execution of the image, the subprocess is deleted. Process deletion is described later in this section.

8.3.2 Input, Output, and Error Devices for Subprocesses

When you call the \$CREPRC system service you can provide equivalence names for the logical names SYS\$INPUT, SYS\$OUTPUT, and SYS\$ERROR. These logical name/equivalence name pairs are placed in the process logical name table for the created process.

The following program segment is an example of defining input, output, and error devices for a subprocess. The notes indicate how these devices are used.

```
INSTREAM: -
    .ASCID /SUB_MAIL_BOX/    ; descriptor for input stream
OUTSTREAM: -
    .ASCID /COMPUTE_OUT/     ; descriptor for output
                                ; and error stream
PROGNAME: -
    .ASCID /COMPUTE.EXE/     ; descriptor for image name
    .
    $CREPRC_S -              ; create process
        IMAGE=PROGNAME, -
        INPUT=INSTREAM, - ①
        OUTPUT=OUTSTREAM, - ②
        ERROR=OUTSTREAM ③
```

- ① The **input** argument equates the equivalence name SUB_MAIL_BOX to the logical name SYS\$INPUT. This logical name may represent a mailbox that the calling process previously created with the Create Mailbox and Assign Channel (\$CREMBX) system service. Any input the subprocess reads from the logical device SYS\$INPUT will be read from the mailbox.
- ② The **output** argument equates the equivalence name COMPUTE_OUT to the logical name SYS\$OUTPUT. All messages the program writes to the logical device SYS\$OUTPUT will be written to this file.
- ③ The **error** argument equates the equivalence name COMPUTE_OUT to the logical name SYS\$ERROR. All system-generated error messages will be written into this file. Because this is the same file as that used for program output, the file effectively contains a complete record of all output produced during the execution of the program image.

Process Control Services

Process Creation

The \$CREPRC system service does not provide default equivalence names for the logical names SYS\$INPUT, SYS\$OUTPUT, and SYS\$ERROR. If none are specified, entries in the group or system logical name tables, if any, may provide equivalences. If, while the subprocess executes, it reads or writes to one of these logical devices and no equivalence name exists, an error condition results.

In a program that creates a subprocess, you can cause the subprocess to share the input, output, or error device of the creating process. The following steps are required:

- Use the Get Device/Volume Information (\$GETDVI) system service to obtain the device name for the logical name SYS\$INPUT, SYS\$OUTPUT, or SYS\$ERROR.
- Specify the address of this descriptor when you specify the **input**, **output**, or **error** argument to the \$CREPRC system service.

This procedure is illustrated in the following example.

```

$PRCDEF
$DVIDEF
DVILIST:
    .WORD    64                ; begin $GETDVI item list
    .WORD    DVI$_DEVNAM      ; maximum of 16 bytes long
    .ADDRESS -                ; get terminal name
        TERM                ; destination of terminal name
    .ADDRESS -
        TERMDESC            ; destination of length of string
    .LONG    0                ; end item list
;
TERMDESC:
    .WORD    64                ; descriptor for terminal name
    .WORD    0                ; maximum of 16 bytes long
;
TERMADDR:
    .ADDRESS -
        TERM
;
TERM:    .BLKB  64                ; terminal name is placed here
;
LOGNAM:  .ASCID  /SYS$INPUT/
;
IMAGENAME:
    .ASCID  /WRKD$: [ORANGE]MIRROR/ ; image for subprocess
;
;
; Determine terminal name
;
$GETDVI_S -
    DEVNAM=LOGNAM, -          ; return information on SYS$INPUT
    ITMLST=DVILIST           ; address of the item list
;
    BLBC    RO,SSERR          ; if not success, go to error routine
;
10$:    $CREPRC_S -           ; create subprocess
        IMAGE=IMAGENAME,      ; running MIRROR
        INPUT=TERMDESC, -     ; using creating process's
        OUTPUT=TERMDESC, -    ; terminal as the input,
        ERROR=TERMDESC, -     ; output, and error device
        BASPRI=#4             ; set base priority to 4

```

Process Control Services

Process Creation

When the subprocess executes, the logical names SYS\$INPUT, SYS\$OUTPUT, and SYS\$ERROR are equated to the device name of the creating process's logical input device. The subprocess can then do one of the following:

- Use VAX RMS to open the file for reading and/or writing
- Use the Assign I/O Channel (\$ASSIGN) system service to assign an I/O channel to the device for input/output operations

In the following example, the program assigns a channel to the device specified by the logical name SYS\$OUTPUT. For example:

```
OUTPUT: .ASCID /SYS$OUTPUT/      ; logical name descriptor
OUTCHAN: .BLKW 1                  ; channel number of output device
      .
      .
      $ASSIGN_S -
          DEVNAM=OUTPUT, -
          CHAN=OUTCHAN
```

For more information on channel assignment for I/O operations, see Section 7, Input/Output Services.

8.3.3 Disk and Directory Defaults for Created Processes

When you use the \$CREPRC system service to create a process to execute an image, the system locates the image file in the default device and directory of the created process. Any created process inherits the current default device and directory of its creator.

If a created process runs an image that is not in its default directory, you must identify the directory and, if necessary, the device in the file specification of the image to be run.

There is no way to define a default device and/or directory for the created process that is different from that of the creating process in a call to \$CREPRC. The created process can, however, define an equivalence for the logical device SYS\$DISK by calling the Create Logical Name (\$CRELNM) system service.

If the process is a subprocess, you can define an equivalence name in the group logical name table, job logical name table or any logical name table shared between the creating process and the subprocess. The created process can also set its own default directory by calling the VAX RMS default directory control routine, SYS\$SETDDIR.

A process can create a process with a default directory that is different from its own by doing the following:

- 1 The process that is creating a new process makes a call to SYS\$SETDIR to change its own default directory.
- 2 The creating process makes a call to \$CREPRC to create the new process.
- 3 The creating process makes a call to SYS\$SETDIR to change its own default directory back to the default directory it had before the first call to SYS\$SETDIR.

Process Control Services

Process Creation

The creating process now has its original default directory. The new process has the different default directory that the creating process had when it created the new process. For details on how to call `SYSS$SETDIR` see the *VAX Record Management Services Reference Manual*.

8.3.4 Controlling Resources of Created Processes

Ordinarily, when you create a subprocess you need only assign it an image to execute and, optionally, the `SYSS$INPUT`, `SYSS$OUTPUT`, and `SYSS$ERROR` devices. The system provides default values for the process's privileges, resource quotas, execution modes, and priority. In some cases, however, you may want to specifically define these values. The arguments to the `$CREPRC` system service that control these characteristics are listed below, with considerations for their use. For details, see the descriptions of arguments to the `$CREPRC` system service in Part II.

- **prvadr**—This argument defines the privilege list for the created process. If this argument is not specified, the privileges of the calling process are used. If the **prvadr** argument is specified, only the privileges specified in the bit mask are used; the privileges of the calling process are not used. For example, a creating process has the user privileges `GROUP` and `TMPMBX`. It creates a process, specifying the user privilege `TMPMBX`. The created process will receive only the user privilege `TMPMBX`; it will not have the user privilege `GROUP`.

If you need to create a process that has a special privilege, note that you must have the user privilege `SETPRV` to create a process with a privilege you do not have.

Symbols associated with privileges are defined by the `$PRVDEF` macro. Each symbol begins with `PRV$V_` and identifies the bit number that must be set to specify a given privilege. The following example shows the data definition for a mask specifying the `GRPNAM` and `GROUP` privileges.

```
PRVMSK: .LONG <10PRV$V_GRPNAM>|<10PRV$V_GROUP> ; grpnam and group
        .LONG 0 ; quadword mask required. no bits set in
                ; high-order longword for these privileges.
```

- **quota**—This argument defines the quota list for a subprocess. Since a subprocess receives a portion of its creator's quotas for timer queue entries, I/O buffers, and so on, you may want to control how much of each quota is assigned to the subprocess. If you do not specify this argument, the system defines default quotas for the subprocess; however, if your process has only the default quotas, you must specify this argument to prevent the subprocess from exhausting all the process's deductible quotas.
- **stsflg**—This argument defines the status flag, a set of bits that control some execution characteristics of the created process, including resource wait mode and process swap mode.
- **baspri**—This argument sets the base execution priority for the created process. If not specified, it defaults to 2 for `VAX MACRO` and `VAX BLISS-32` and to 0 for other languages. If you want a subprocess to have a higher priority than its creator, you must have the user privilege `ALTPRI` to raise the priority level.

8.3.5 Detached Processes

The creation of a detached process is primarily a function performed by VAX/VMS at login time. The DETACH privilege controls the ability to create a detached process with a UIC that is different from the UIC of the creating process. The **uic** argument to the \$CREPRC system service provides one way to define whether a process is a subprocess or a detached process; it provides the created process with a user identification code (UIC). If you omit the UIC argument, the \$CREPRC system service creates a subprocess that executes under UIC of the creating process.

You can also create a detached process with the same UIC as the creating process by specifying the detach flag in the **stsf** argument. You do not need DETACH privilege to create a detached process with the same UIC as the creating process.

8.4 Interprocess Control and Communication

Processes can be either wholly independent or cooperative. The sections that follow discuss considerations for developing applications that require the concurrent execution of many programs.

8.4.1 Restrictions on Process Creation and Control

There are three levels of process control privilege.

- 1 Processes with the same UIC can always issue process control services for one another.
- 2 The GROUP privilege is required to issue process control services for other processes executing in the same group.
- 3 The WORLD privilege is required to issue process control services for any process in the system.

Additional privileges are required to perform some specific functions, for example, to set a process's base priority to a higher level than that of the requestor.

8.4.2 Process Identification

There are two types of process identification.

- 1 Process identification number (PID). The system assigns this unique 32-bit number to a process when it is created. If you provide the **pidadr** argument to the \$CREPRC system service, the system returns the process identification number at the location specified. You can then use the process identification number in subsequent process control services.
- 2 Process name. A process name is a 1- through 15-character text name string. Each process name must be unique within its group (processes in different groups can have the same name). You can assign a name to a process by specifying the **prcnam** argument when you create it. You can then use this name to refer to the process in other system service calls. Note that you cannot use a process name to specify a process outside the caller's group; you must use a process identification number.

Process Control Services

Interprocess Control and Communication

In the examples shown in the preceding sections, the subprocesses are not identified.

If you want to control the execution of a subprocess, you must give it a name. You must also name detached processes that execute in the same group if they communicate with each other or issue control functions affecting each other.

For example, you might call the \$CREPRC system service as follows:

```
ORION: .ASCID /ORION/      ; descriptor for process name
ORIONID:
        .LONG  0          ; process id returned
        .
        .
        $CREPRC_S -
            PRCNAM=ORION, -
            PIDADR=ORIONID,...
```

The service returns the process identification in the longword at ORIONID. You can now use either the process name (ORION) or the process identification (ORIONID) to refer to this process in other system service calls.

A process can set or change its own name with the Set Process Name (\$SETPRN) system service. For example, a process can set its name to CYGNUS as follows:

```
CYGNUS: .ASCID /CYGNUS/    ; descriptor for process name
        .
        .
        $SETPRN_S -
            PRCNAM=CYGNUS
```

Most of the process control services accept either the **prcnam** or **pidadr** arguments, or both. However, for the following reasons it is better to identify a process by its process identification number:

- The service executes faster because it does not have to search a table of process names.
- You must use the process identification number for a process not in your group (see Section 8.4.2.1).

If neither the process name argument nor the process identification number argument is specified, the service is performed for the calling process. Table 8-1 gives a summary of the possible combinations of these arguments and an explanation of how the services interpret them.

8.4.2.1 Process Naming Within Groups

Process names are always qualified by their group number. The system maintains a table of all process names and the UIC associated with each process name. When the **prcnam** argument is used in a process control service, the table is searched for an entry that contains the specified process name and the group number of the calling process.

To use process control services on processes within its group, a calling process must have the user privilege GROUP; this privilege is not required when specifying a process with the same UIC as the caller.

Process Control Services

Interprocess Control and Communication

The search for a process name fails if the specified process name does not have the same group number as the caller. The search fails even if the calling process has the user privilege WORLD. To execute a process control service for a process that is not in the caller's group, the requesting process must use a process identification and must have the user privilege WORLD.

Table 8-1 Process Identification

Process Name Specified?	Process ID Address Specified?	Process ID Contains:	Resultant Action by Services
No	No	—	The process identification of the calling process is used. The process identification is not returned.
No	Yes	0	The process identification of the calling process is used and returned.
No	Yes	Process id	The process identification is used and returned.
Yes	No	—	The process name is used. The process identification is not returned.
Yes	Yes	0	The process name is used and the process identification is returned.
Yes	Yes	Process id	The process identification is used and returned; the process name is ignored.

8.4.2.2

Obtaining Information About Processes

The Get Job/Process Information (\$GETJPI) system service allows a process to obtain information about itself or another process. For complete details about the \$GETJPI system service, see the description of \$GETJPI in Part II.

8.4.2.3

Techniques for Interprocess Communication

Processes can communicate in the following ways:

- Common event flag clusters
- Logical names
- Mailboxes
- Global sections
- Lock Management system services
- Files

Each communication technique offers different possibilities in terms of the speed at which it communicates information and the amount of information it can communicate. For example, files offer the possibility of sharing an effectively limitless amount of information; the files technique is the slowest of the techniques because the disk must be accessed to share information.

Process Control Services

Interprocess Control and Communication

Like files, global sections offer the possibility of sharing large amounts of information. Because sharing information through global sections requires only memory access, it is the fastest communication technique.

Logical names and mailboxes can communicate moderate amounts of information. Because each technique operates through a relatively complex system service, they are faster than files, but slower than the other communication techniques.

The lock management services and common event flag cluster techniques can communicate relatively small amounts of information. With the exception of global sections, they are the fastest of the interprocess communication techniques.

Common Event Flag Clusters: Processes executing within the same group can use common event flag clusters to signal the occurrence or completion of particular activities. For details on event flags, event flag clusters, and an example of how cooperating processes in the same group use a common event flag, see Section 4, Event Flag Services.

Logical Name Tables: Processes executing in the same job can use the job-wide logical name table to provide member processes with equivalence names for logical names. Processes executing in the same group can use the group logical name table. A process must have the user privilege GRPNAM to place names in the group logical name table. All processes in the system can use the system logical name table. Processes can also create and use user-defined logical name tables. For details on logical names and logical name tables, see Section 6, Logical Name Services.

Mailboxes: Mailboxes can be used as virtual input/output devices to pass information, messages, or data among processes. For details on how to create and use mailboxes, with an example of how cooperating processes use a mailbox, see Section 7, Input/Output Services. Mailboxes may also be used to provide a creating process with a way to determine when and under what condition a created subprocess was deleted. See Section 8.7.2 for an example of a termination mailbox.

Global Sections: Global sections can be either disk files or pagefile sections containing shareable code or data. Through the use of memory management services, these files can be mapped to the virtual address space of more than one process. In the case of a data file on disk, cooperating processes can synchronize reading and writing the data in physical memory; as data is updated, system paging results in the updated data being written directly back into the disk file. Global pagefile sections are useful for temporary storage of common data; they are not mapped to a disk file, instead they simply page to the system default page file. Global sections are described in more detail in Section 11.6.

Lock Management System Services: Processes can use the lock management system services to control access to resources (any entity on the system that the process can read, write, or execute). In addition to controlling access, the lock management services provide a mechanism for passing information among processes that have access to a resource (lock value blocks). Blocking ASTs can be used to notify a process that other processes are waiting for a resource. For information on the lock management system services, see Section 12, Lock Management System Services.

8.5 Process Hibernation and Suspension

There are two ways to temporarily halt the execution of a process: hibernation, performed by the Hibernate (\$HIBER) system service and suspension, performed by the Suspend Process (\$SUSPND) system service. The process can continue execution normally only after a corresponding Wake from Hibernation (\$WAKE) system service, if it is hibernating, or after a Resume Process (\$RESUME) system service, if it is suspended.

Process hibernation and suspension are compared in Table 8-2.

Table 8-2 Process Hibernation and Suspension

Hibernation	Suspension
Can only cause self to hibernate	Can suspend self or another process, depending on privilege
Reversed by \$WAKE system service	Reversed by \$RESUME system service
Interruptible; can receive ASTs	Noninterruptible; cannot receive ASTs
Can wake self	Cannot cause self to resume
Can schedule wakeup at an absolute time or at a fixed time interval	Cannot schedule resumption
Requires little system overhead	Requires system dynamic memory

8.5.1 Process Hibernation

The hibernate/wake mechanism provides an efficient way to prepare an image for execution and then place it in a wait state until it is needed. When the wake request is issued, the image is reactivated with little delay or system overhead.

For example, if you create a subprocess that must execute the same function repeatedly and must execute immediately when it is needed, you could use the \$HIBER and \$WAKE system services as shown in the following example.

A variation of the \$WAKE system service schedules a wakeup for a hibernating process at a fixed time or at an elapsed (delta) time interval. This is the Schedule Wakeup (\$SCHDWK) system service. Using the \$SCHDWK service, a process can schedule a wakeup for itself before issuing a \$HIBER call. For an example of how to use the \$SCHDWK system service, see Section 8, Timer and Time Conversion Services.

Hibernating processes can be interrupted by Asynchronous System Traps (ASTs), as long as AST delivery is enabled. The process can call \$WAKE on its own behalf in the AST service routine, and continue execution following the execution of the AST service routine. For a description of ASTs and how to use them, see Section 5, AST (Asynchronous System Trap) Services.

Process Control Services

Process Hibernation and Suspension

```

Process TAURUS
ORION: .ASCID /ORION/           ; descriptor for subprocess name
FASTCOMP: .ASCID /COMPUTE.EXE/ ; descriptor for image name

① $CREPRC_S -                   ; create ORION
      PRCNAM=ORION, "
      IMAGE=FASTCOMP,...
      BSBW  ERROR                ; continue

② $WAKE_S PRCNAM=ORION          ; wake ORION
      BSBW  ERROR

      $WAKE_S PRCNAM=ORION      ; wake ORION again
      BSBW  ERROR

Process ORION
.ENTRY COMPUTE,~M<> ③; entry mask
10$: $HIBER_S          ; sleep
      BSBW  ERROR
      .
      .
      .
      BRW  10$          ; back to sleep

```

- ① Process TAURUS creates the process ORION, specifying the descriptor for the image named COMPUTE.
- ② The image COMPUTE is initialized, and ORION issues the \$HIBER system service.
- ③ At an appropriate time, TAURUS issues a \$WAKE request for ORION. ORION continues execution following the \$HIBER service call. When it finishes its job, ORION loops back to repeat the \$HIBER call and to wait for another wakeup.

8.5.2 Alternate Methods of Hibernation

You can use two additional techniques to cause a process to hibernate.

- Specify the **stsf** argument for the \$CREPRC system service, setting the bit that requests \$CREPRC to place the created process in a state of hibernation as soon as it is initialized.
- Specify the /DELAY, /SCHEDULE, or /INTERVAL qualifier to the RUN command when you execute the image from the command stream.

When you use the first method, the creating process can control when to wake the created process. When you use the RUN command, the qualifiers listed above control when the process will be awakened.

If the /INTERVAL qualifier is used and the image to be executed does not call the \$HIBER system service, the image is placed in a state of hibernation whenever it issues a RET instruction. Each time the image is reawakened, it begins executing at its entry point. If the image does call \$HIBER, each time it is awakened it begins executing at either the point following the call to \$HIBER or at its entry point (if it last issued a RET instruction).

Process Control Services

Process Hibernation and Suspension

If wakeup requests are scheduled at time intervals, the image can be terminated with the Delete Process (\$DELPRC) or Force Exit (\$FORCEX) system service, or from the command level with the STOP command. The \$DELPRC and \$FORCEX system services are described later in this section. The RUN and STOP commands are described in the *VAX/VMS DCL Dictionary*.

These techniques allow you to write programs that can be executed once, on request, or cyclically. Note that the program must ensure the integrity of data areas that are modified during its execution, as well as the status of opened files.

8.5.3 Suspension

Using the Suspend Process (\$SUSPND) system service, a process can place itself or another process into a wait state similar to hibernation. Suspension, however, is a more pronounced state of hibernation. VAX/VMS provides no system service to force a process to be swapped out, but the \$SUSPND system service can accomplish the task. Suspended processes are the first processes to be selected for swapping. A suspended process cannot be interrupted by ASTs, and can resume execution only after another process issues a Resume Process (\$RESUME) system service for it. If ASTs were queued for the process while it was suspended, they are delivered when the process resumes execution. This is an effective tool for blocking delivery of all ASTs.

8.6 Image Exit

When image execution completes normally, the operating system performs a variety of image run-down functions. If the image was executed by the command interpreter, image run-down prepares the process for the execution of another image. If the image was not executed by the command interpreter—for example, if it was executed by a subprocess—the process is deleted.

These exit activities are also initiated when an image completes abnormally, as a result of any of the following conditions:

- Specific error conditions caused by improper specifications when a process was created. For example, if an invalid device name is specified for the SYS\$INPUT, SYS\$OUTPUT, or SYS\$ERROR logical name, or if an invalid or nonexistent image name is specified, the error condition is signalled in the created process.
- An exception occurring during execution of the image. When an exception occurs, any user-specified condition handlers receive control to handle the exception. If there are no user-specified condition handlers, a system-declared condition handler receives control, and it initiates exit activities for the image. Condition handling is described in Section 9, Condition-Handling Services.
- A Force Exit (\$FORCEX) system service issued on behalf of the process by another process.

Process Control Services

Image Exit

8.6.1 Image Run-down Activities

The operating system performs image run-down functions that release system resources obtained by a process while it executed in user mode. These activities and the order in which they occur are listed below.

- 1 Any outstanding I/O requests on the I/O channels are canceled and I/O channels are deassigned.
- 2 Memory pages occupied or allocated by the image are deleted and the process's working set size limit is readjusted to its default value.
- 3 All devices allocated to the process at user mode are deallocated (devices allocated from the command stream in supervisor mode are not deallocated).
- 4 Timer-scheduled requests, including wake-up requests, are canceled.
- 5 Common event flag clusters are disassociated.
- 6 User mode ASTs that are queued but have not been delivered are deleted, and ASTs are enabled for user mode.
- 7 Exception vectors declared in user mode, compatibility mode handlers, and change mode to user handlers are reset.
- 8 System service failure exception mode is disabled.
- 9 All process private logical names and logical name tables created for user mode are deleted. Deletion of a logical name table causes all names in that table to be deleted. Note that names entered in shareable logical name tables such as the job or group table are not deleted at image run-down, regardless of the access mode for which they were created.

8.6.2 The \$EXIT System Service

To initiate the run-down activities described above, the system calls the Exit (\$EXIT) system service on behalf of the process. In some cases, a process can call \$EXIT to terminate the image itself, for example, if an unrecoverable error occurs.

The \$EXIT system service accepts a status code as an argument. If you use \$EXIT to terminate image execution, you can use this status code argument to pass information about the completion of the image. If an image returns without calling \$EXIT, the current value in R0 is passed as the status code when the system calls \$EXIT.

This status code is used as follows:

- The command interpreter uses the status code to optionally display an error message when it receives control following image run-down.
- If the image has declared an exit handler, the status code is written in the address specified in the exit control block.
- If the process was created by another process, and the creator has specified a mailbox to receive a termination message, the status code is written into the termination mailbox when the process is deleted.

8.6.3 Exit Handlers

Exit handlers are procedures that can perform image-specific cleanup or run-down operations. For example, if an image uses memory to buffer data, an exit handler can ensure that the data is not lost when the image exits as the result of an error condition.

To establish an exit-handling routine, you must set up an exit control block and specify the address of the control block in the call to the Declare Exit Handler (\$DCLEXH) system service. Exit handlers are called using standard calling conventions; you can provide arguments to the exit handler in the exit control block. The first argument in the control block argument list must specify the address of a longword for the system to write the status code from \$EXIT.

If an image declares more than one exit handler, the control blocks are linked together on a last-in, first-out basis. After an exit handler has been called and returns control, the control block is removed from the list. Exit control blocks can also be removed prior to image exit with the Cancel Exit Handler (\$CANEXH) system service.

Exit handlers can be declared from system routines executing in supervisor or executive modes. These exit handlers are also linked together in other lists, and receive control after exit handlers declared from user mode have been executed.

Exit handlers are called as a part of the \$EXIT system service. While a call to the \$EXIT system service often precedes image run-down activities, it is not a part of image run-down. There is no guaranteed way to insure that exit handlers will be called if an image terminates in a nonstandard way, such as, a \$DELPRC call from another process.

8.6.4 Forced Exit

The Force Exit (\$FORCEX) system service provides a way for a process to initiate image run-down for another process. For example, the following call to \$FORCEX causes the image executing in the process CYGNUS to exit.

```
CYGNUS: .ASCID /CYGNUS/      ; process name descriptor
```

```
$FORCEX_S -  
PRCNAM=CYGNUS
```

Because the \$FORCEX system service calls the \$EXIT system service, any exit handlers that are declared for the image are executed before image run-down. Thus, if the process is using the command interpreter, the process is not deleted, and can run another image. Because the \$FORCEX system service uses the AST mechanism, an exit cannot be performed if the process being forced to exit has disabled the delivery of ASTs. AST delivery, and how it is disabled and reenabled, is described in Section 5.

The following program segment shows an example of an exit-handling routine.

Process Control Services

Image Exit

```
EXITBLOCK: ①
    .LONG 0
    .ADDRESS -
    .LONG EXITRTN
    .ADDRESS -
    STATUS
STATUS: .BLKL 1
    .ENTRY PEGASUS,~M<R2,R3>
    ② $DCLEXH_S -
    DESBLK=EXITBLOCK
    BSBW ERROR
    RET
; exit handler
    .ENTRY EXITRTN,~M<R2>
    ③ BLBS STATUS,10$
10$: RET
```

; exit control block
; system uses this for pointer
; address of exit handler
; number of args for handler
; destination of status code
; status code from \$EXIT
; entry mask for PEGASUS
; declare exit handler
; end of main routine
; entry mask
; normal exit? yes, finish
; no, clean up
; finished

- ① EXITBLOCK is the exit control block for the exit handler EXITRTN. The third longword indicates the number of arguments to be passed. In this example, only one argument is passed, the address of a longword for the system to store the return status code. This argument must be provided in an exit control block.
- ② The \$DCLEXH system service call designates the address of the exit control block, thus declaring EXITRTN as an exit handler.
- ③ EXITRTN checks the status code. If this is a normal exit, EXITRTN returns control. Otherwise, it handles the error condition.

8.7 Process Deletion

Process deletion completely removes a process from the system. A process can be deleted by any of the following events:

- The Delete Process (\$DELP RC) system service is called.
- A process that created a subprocess is deleted.
- An interactive process uses the DCL command LOGOUT.
- A batch job reaches the end of its command file.
- An interactive process uses the DCL command STOP/ID=p id or STOP username.
- A process that contains a single image calls the Exit (\$EXIT) system service.

No process can be deleted until any subprocesses it has created have been deleted. When the system is called to delete a process as a result of any of the above conditions, it first locates all subprocesses, searching hierarchically.

The lowest subprocess in the hierarchy is a subprocess that has no descendent subprocesses of its own. When that subprocess is deleted, its parent subprocess becomes a subprocess that has no descendent subprocesses and it can be deleted. The topmost process in the hierarchy is the process that is the ultimate parent process of all the other subprocesses.

Beginning with the lowest process in the hierarchy and completing with the topmost process, each of the following procedures is performed.

- The image executing in the process is run down. The image run-down that occurs during process deletion is the same as that described in Section 8.6.1. When a process is deleted, however, the run-down releases all system resources, including those acquired from access modes other than user mode.
- Resource quotas are released to the creating process, if the process being deleted is a subprocess.
- If the creating process specified a termination mailbox, a message indicating that the process being deleted is sent to the mailbox. For detached processes created by the system, the termination message is sent to the system job controller.
- The control region of the process's virtual address space is deleted. (The control region consists of memory allocated and used by the system on behalf of the process.)
- All system-maintained information about the process is deleted.

8.7.1 The Delete Process System Service

A process can delete itself or another process at any time, depending on the restrictions outlined in Section 8.4.1. The Delete Process (\$DELPRC) system service deletes a process. For example, if a process has created a subprocess named CYGNUS, it can delete CYGNUS as shown below:

```
CYGNUS: .ASCID /CYGNUS/ ;DESCRIPTOR FOR PROCESS NAME
.
.
$DELPRC_8-
PRCNAM=CYGNUS
```

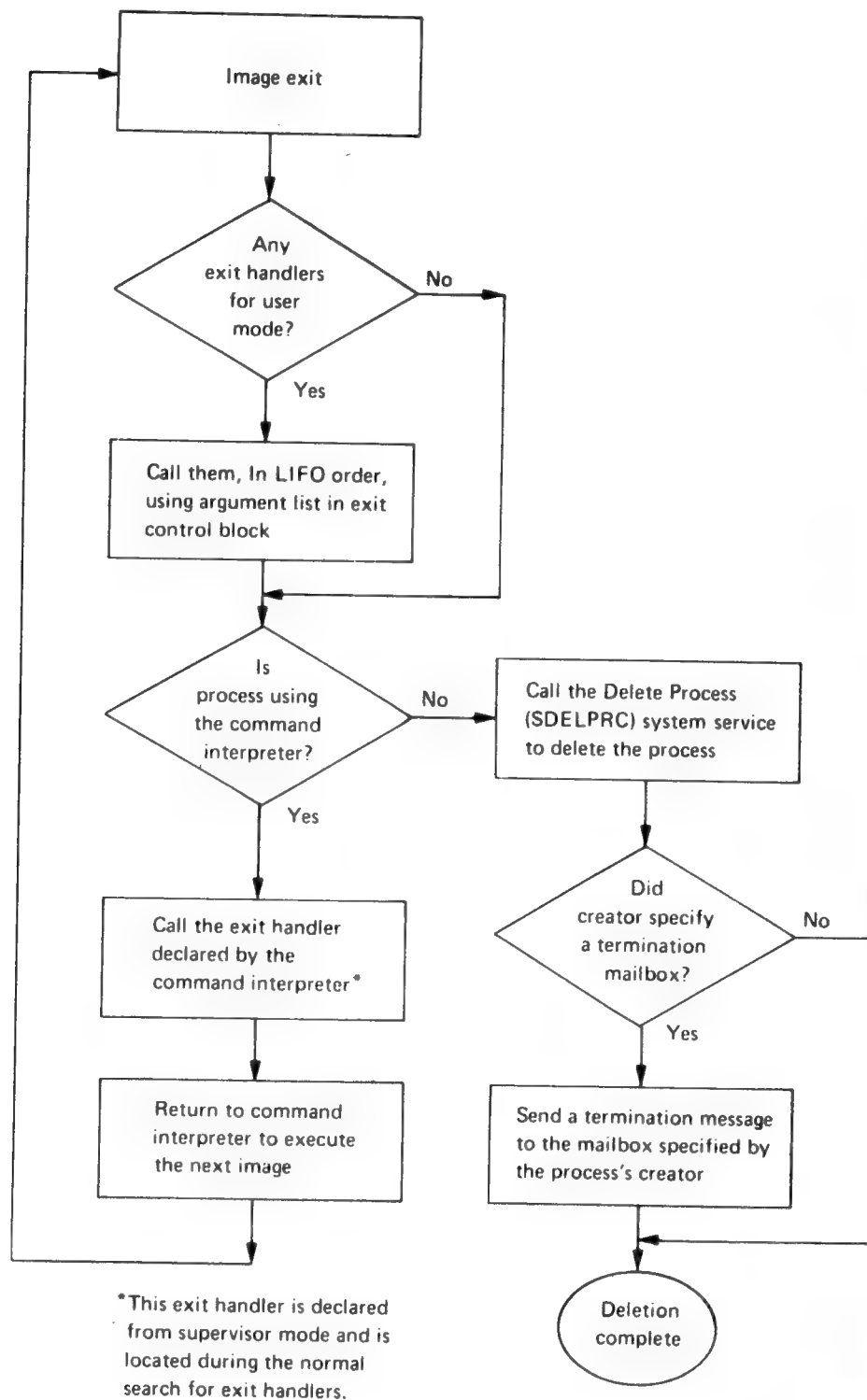
Because a subprocess is automatically deleted when the image it is executing terminates (or when the command stream for the command interpreter reaches end-of-file), you do not normally need to issue the \$DELPRC system service explicitly.

As an alternative to deleting a process to stop an image, you can use the Force Exit (\$FORCEX) system service to force the exit of the image executing in a process (see Section 8.6.4).

Process Control Services

Process Deletion

Figure 8-1 Image Exit and Process Deletion



ZK-857-82

8.7.2 Termination Mailboxes

A termination mailbox provides a process with a way of determining when, and under what conditions, a process that it has created is being deleted. The Create Process (\$CREPRC) system service accepts the unit number of a mailbox as an argument. When the created process is deleted, the mailbox receives a termination message.

The first word of the termination message contains the symbolic constant, MSG\$_DELPROC, which indicates that it is a termination message. The second longword of the termination message contains the final status value of the image. The remainder of the message contains system accounting information used by the job controller, and is in fact identical to the first part of the accounting record sent to the system accounting log file. The complete format of the termination message is provided with the description of the \$CREPRC system service in Part II.

The creating process can, if necessary, determine the process identification of the process being deleted from the I/O status block posted when the message is received in the mailbox. The second longword of the IOSB contains the process identification of the process that is being deleted.

A termination mailbox cannot be located in memory shared by multiple processors.

The example below illustrates a complete sequence of process creation, with a termination mailbox. The Create Mailbox and Assign Channel (\$CREMBX), Get Device/Volume Information (\$GETDVI), and Queue I/O Request (\$QIO) system services are described in greater detail in Section 7, Input/Output Services.

```

EXCHAN:
    .BLKW 1 ; to hold channel number of mailbox
MBXINFO:
    .WORD 4 ; start $GETDVI item list
    .WORD 4 ; length of buffer
    .WORD DVI$_UNIT
    .ADDRESS -
        UNITNUM ; address of buffer
    .LONG 0 ; no return length needed
    .LONG 0 ; end item list
UNITNUM:
    .WORD ; to receive unit number
;
EXITMSG:
    .BLKB ACC$_K__TERMLEN ; buffer for mailbox message
    ; (see $SENDACC explanation
    ; for ACC$_K__TERMLEN)
MBXIOSB:
    .BLKW 1 ; quadword I/O status block
MBLEN: .BLKW 1 ; length of I/O
MBPID: .BLKL 1 ; receives PID of process deleted
LYRAPID:
    .LONG 0 ; get PID of subprocess
LYREXE: .ASCID /LYRA.EXE/ ; name of image for subprocess
    
```

Process Control Services

Process Deletion

```

1      $CREMBX__S -           ; create mailbox
      CHAN=EXCHAN, -
      MAXMSG=_$84, -
      PROMSK=_$0, -
      BUFQUO=_$240
      BSBW ERROR
2      $GETDVI__S -          ; get mailbox info
      CHAN=EXCHAN, -
      ITMLST=MBXINFO
      BSBW ERROR
3      $CREPRC__S -          ; create subprocess
      IMAGE=LYREXE, -
      PIDADR=LYRAPID, -
      MBXUNT=UNITNUM ; specify termination mailbox
4      BSBW ERROR
      $QIO__S CHAN=EXCHAN, - ; QIO (read) to mailbox
      FUNC=_$IO$__READVBLK, -
      ASTADR=EXITAST, -
      IOSB=MBXIOSB, -
      P1=EXITMSG, -
      P2=_$ACC$K__TERMLEN
      BSBW ERROR
      ; continue execution
      RET
; AST routine for termination message
5      .ENTRY EXITAST,_M<> ; entry mask
      CMPW MBXIOSB,_$SS$__NORMAL ; I/O successful?
      BNEQ 20$ ; branch if not
      CMPW EXITMSG+ACC$W__MSGTYP,_$MSG$__DELPROC
      ; is it a termination msg?
      BNEQ 20$ ; no, something else
      CMPL LYRAPID,MBPID ; is it LYRA?
      BNEQ 20$ ; no, somebody else
      CMPL EXITMSG+ACC$L__FINALSTS,_$SS$__NORMAL
      ; deleted normally?
      BEQL 10$ ; yes, return
      ; no, respond to error in LYRA
10$: RET ; AST routine finished
20$: ; handle all other conditions

```

- 1 The Create Mailbox and Assign Channel (\$CREMBX) system service creates the mailbox, and returns the channel number at EXCHAN.
- 2 The item list for the Get Device/Volume Information (\$GETDVI) system service specifies that the unit number of the mailbox is to be returned.
- 3 The Create Process (\$CREPRC) system service creates a process to execute the image LYRA.EXE, and returns the process identification at LYRAPID. The **mbxunt** argument refers to the unit number of the mailbox, obtained from the Get Device/Volume Information (\$GETDVI) system service.
- 4 The Queue I/O Request queues a read request to the mailbox, specifying an AST service routine to receive control when the mailbox receives a message and the address of a buffer to receive the message. The information in the message can be accessed by the symbolic offsets defined in the \$ACCDDEF macro. The process continues executing.

- When a message is received in the mailbox, the AST service routine, EXITAST, receives control. Because this mailbox can be used for other interprocess communication, the AST routine checks:
 - For successful completion of the I/O operation by examining the first word in the IOSB
 - That the message received is a termination message by examining the message type field in the termination message at the offset ACC\$W_MSGTYPE
 - The process identification of the process that has been deleted by examining the second longword of the IOSB
 - The completion status of the process by examining the status field in the termination message at the offset ACC\$L_FINALSTS

In this example, the AST service routine performs special action when the subprocess is deleted. All other messages or error conditions cause a branch to the label 20\$.

8.8 Example of Using Process Control Services

```

      INSTALLED.FOR
! Installed common to be linked with INCOME.FOR and
! CALC_TAXES.FOR.
! Unless the shareable image created from this file is
! in SYS$SHARE, you must define a group logical name
! INSTALLED and equivalence it to the full file specification
! of the shareable image.
INTEGER*4 INCOME (200),
2         TAXES (200),
2         NET (200)
COMMON /CALC/ INCOME,
2         TAXES,
2         NET
END

      INCOME.FOR
! status and system routines
INCLUDE '($SDEF)'
INCLUDE '($IODEF)'
INTEGER STATUS,
2         LIB$GET_LUN,
2         LIB$GET_EF,
2         SYS$CLREF,
2         SYS$CREMBX,
2         SYS$CREPRC,
2         SYS$GETDVIW,
2         SYS$QIO,
2         SYS$WAITFR
! set up for SYS$GETDVI
INTEGER*4 UNIT_BUF,
2         UNIT_LEN
INTEGER*2 UNIT_BUF_LEN,
2         UNIT_BUF_CODE
INTEGER*4 UNIT_BUF_ADDR,
2         UNIT_LEN_ADDR,
2         END_LIST /0/
EXTERNAL DVI$UNIT
COMMON /GETDVI_LIST/ UNIT_BUF_LEN,
2         UNIT_BUF_CODE,
2         UNIT_BUF_ADDR,
2         UNIT_LEN_ADDR,
2         END_LIST

```

Process Control Services

Example of Using Process Control Services

```
! name and I/O channel for mailbox
CHARACTER*(*) MBX_NAME
PARAMETER (MBX_NAME = 'MAIL_BOX')
INTEGER*2 MBX_CHAN
!logical unit number for I/O
INTEGER*4 MBX_LUN
!mailbox message
CHARACTER*84 MBX_MESSAGE
INTEGER*4 READ_CODE,
2      LENGTH
! I/O status block
INTEGER*2 IOSTAT,
2      MSG_LEN
INTEGER*4 READER_PID
COMMON /IOBLOCK/ IOSTAT,
2      MSG_LEN,
2      READER_PID

! declare calculation variables in installed common
INTEGER*4 INCOME (200),
2      TAXES (200),
2      NET (200)
COMMON /CALC/ INCOME,
2      TAXES,
2      NET

! flag to indicate taxes calculated
INTEGER*4 TAX_DONE

! get and clear an event flag
STATUS = LIB$GET_EF (TAX_DONE)
IF (.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))
STATUS = SYS$CLREF (%VAL(TAX_DONE))
IF (.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))

!create the mailbox
STATUS = SYS$CREMBX (,
2      MBX_CHAN,
2      ,
2      MBX_NAME)
IF (.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))

! get unit number of the mailbox
UNIT_BUF_LEN = 4
UNIT_BUF_CODE = %LOC(DVI$UNIT)
UNIT_BUF_ADDR = %LOC(UNIT_BUF)
UNIT_LEN_ADDR = %LOC(UNIT_LEN)
STATUS = SYS$GETDVIW (,
2      %VAL(MBX_CHAN),
2      MBX_NAME,          ! device
2      UNIT_BUF_LEN,      ! common
2      ...)
IF (.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))

! create subprocess to calculate taxes
STATUS = SYS$CREPRC (,
2      'CALC_TAXES', !image
2      ,
2      'CALC_TAXES', !process name
2      %VAL(4),      !priority
2      ,
2      %VAL(UNIT_BUF),)
IF (.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))

! Asynchronous read to termination mailbox
! sets flag when tax calculations complete.
READ_CODE = IO$READVBLK
LENGTH = 84
STATUS = SYS$QIO (%VAL(TAX_DONE), ! indicates read complete
2      %VAL(MBX_CHAN), ! channel
2      %VAL(READ_CODE), ! function code
2      IOSTAT,... ! status block
2      %REF(MBX_MESSAGE),! P1
2      %VAL(LENGTH),...) ! P2
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
```

Process Control Services

Example of Using Process Control Services

```
! calculate incomes
.
.
! wait until taxes are calculated
STATUS = SYS$WAITFR (%VAL(TAX_DONE))
IF (.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))
! check mailbox I/O
IF (.NOT. IOSTAT) CALL LIB$SIGNAL (%VAL(IOSTAT))
! calculate net income after taxes
.
.
END

                                CALC_TAXES.FOR
! declare calculation variables in installed common
INTEGER*4 INCOME (200),
2          TAXES (200),
2          NET (200)
COMMON /CALC/ INCOME,
2            TAXES,
2            NET
! calculate taxes
.
.
END
```

The preceding FORTRAN example calculates gross income and taxes, and then uses the results to calculate net income.

INCOME.FOR uses SYSS\$CREPRC, specifying a termination mailbox, to create a subprocess to calculate taxes (CALC_TAXES) while INCOME calculates gross income. INCOME issues an asynchronous read to the termination mailbox specifying an event flag to be set when the read completes. (The read completes when CALC_TAXES completes terminating the created process and causing the system to write to the termination mailbox.)

After finishing its own gross income calculations, INCOME.FOR waits for the flag that indicates CALC_TAXES has completed and then figures net income.

CALC_TAXES.FOR passes the tax information to INCOME.FOR using the installed common block created from INSTALLED.FOR.

9

Timer and Time Conversion Services

Many applications require the scheduling of program activities based on clock time. Under VAX/VMS, an image can schedule events for a specific time of day or after a specified time interval. The following services are timer and time conversion services.

- Get Time (\$GETTIM)
- Convert Binary Time to Numeric Time (\$NUMTIM)
- Convert Binary Time to ASCII String (\$ASCTIM)
- Convert ASCII String to Binary Time (\$BINTIM)
- Set Timer (\$SETIMR)
- Cancel Timer Request (\$CANTIM)
- Schedule Wakeup (\$SCHDWK)
- Cancel Wakeup (\$CANWAK)
- Set System Time (\$SETIME)

You can use timer services to schedule, convert or cancel events. For example, you may use the timer services to:

- Schedule the setting of an event flag or the queuing of an asynchronous system trap (AST) for the current process, or cancel a pending request that has not yet been honored
- Schedule a wake-up request for a hibernating process, or cancel a pending wake-up request that has not yet been honored
- Set or recalibrate the current system time, if the caller has the proper user privileges

The timer services require you to specify the time in a 64-bit format. To work with the time in different formats, you can use time conversion services to:

- Obtain the current date and time in an ASCII string or in system format
- Convert an ASCII string into the system time format
- Convert a system time value into an ASCII string
- Convert the time from system format to integer values

This section describes the system time format and the services that use it, with examples of scheduling program activities using the timer services.

Timer and Time Conversion Services

The System Time Format

9.1 The System Time Format

VAX/VMS maintains the current date and time in 64-bit format. The time value is a binary number in 100-nanosecond units offset from the system base date and time, which is 00:00 o'clock, November 17, 1858 (the Smithsonian base date and time for the astronomical calendar). Time values must be passed to, or returned from, system services as the address of a quadword containing the time in 64-bit format. A time value can be expressed as either of the following:

- An absolute time that is a specific date and time of day. Absolute times are always positive values (or zero).
- A delta time that is an offset from the current time to a time or date in the future. Delta times are always expressed as negative values.

If you specify zero as the address of a time value, VAX/VMS will supply the current date and time.

9.2 Obtaining the Current Date and Time

The current time can be obtained in system format with the Get Time (\$GETTIM) system service, which places the time into a quadword buffer.

```
TIME: .BLKQ 1 ; buffer for time
.
$GETTIM_S -
TIMADR=TIME ; get time
```

This call to \$GETTIM returns the current date and time in system format in the quadword buffer TIME.

The Convert Binary Time to ASCII String (\$ASCTIM) system service converts a time in system format to an ASCII string and returns the string in a 23-byte buffer, call the \$ASCTIM system service as follows:

```
ATIMENOW: ; descriptor for ASCII time
.LONG 23 ; length of buffer
.ADDRESS -
TIMESTR ; address of buffer
TIME_VALUE: ; 64-bit time value to be converted
.BLKQ 1
TIMESTR:
.BLKB 23 ; 23 bytes returned
.
$ASCTIM_S -
TIMBUF=ATIMENOW, -
TIMADR=TIME_VALUE
```

Because the address of a 64-bit time value was not supplied, the default value, zero, was used.

The string returned by the service has the following format:

```
dd-mmm-yyyy hh:mm:ss.cc
```

where dd is the day of the month, mmm is the month (a three-character alphabetic abbreviation), yyyy is the year, and hh:mm:ss.cc is the time in hours, minutes, seconds, and hundredths of seconds.

Timer and Time Conversion Services

Obtaining an Absolute Time in System Format

9.3

Obtaining an Absolute Time in System Format

The converse of the \$ASCTIM system service is the Convert ASCII String to Binary Time (\$BINTIM) system service. You provide the service with the time in the ASCII format shown above, and the service converts the string to a time value in 64-bit format. You can then use this returned value as input to a timer scheduling service.

When you specify the ASCII string buffer, you can omit any of the fields, and the service uses the current date or time value for the field. Thus, if you want a timer request to be date-independent, you could format the input buffer for the \$BINTIM service as shown below. The two hyphens that are normally embedded in the date field must be included, and at least one blank must precede the time field.

```
ASCII_NOON:
    .ASCID  /-- 12:00:00.00/      ; descriptor for ASCII 12 noon
BINARY_NOON:
    .BLKQ   1                    ; buffer for binary 12 noon
    .
    .
    $BINTIM_S -                  ; convert time
        TIMBUF=ASCII_NOON, -
        TIMADR=BINARY_NOON
```

When the \$BINTIM service completes, a 64-bit time value representing "noon today" is returned in the quadword at BINARY_NOON.

9.4

Obtaining a Delta Time in System Format

The \$BINTIM system service also converts ASCII strings to delta time values to be used as input to timer services. The buffer for delta time ASCII strings has the following format:

dddd hh:mm:ss.cc

The first field, indicating the number of days, must be specified as 0 if you are specifying a delta time for the current day.

The following example shows how to use the \$BINTIM service to obtain a delta time in system format.

```
ATENMIN:
    .ASCID  /0 00:10:00.00/ ; descriptor for ASCII ten minutes
BTENMIN:
    .BLKQ   1                ; buffer for binary ten minutes
    .
    .
    $BINTIM_S -              ; convert time
        TIMBUF=ATENMIN, -
        TIMADR=BTENMIN
```

If you are a VAX MACRO programmer, you can also specify approximate delta time values when you assemble a program, using two MACRO .LONG directives to represent a time value in terms of 100-nanosecond units. The arithmetic is based on the following formula:

1 second = 10 million * 100 nanoseconds

For example, the following statement defines a delta time value of five seconds.

```
FIVESEC: .LONG -10*1000*1000*5,-1 ; five seconds
```

Timer and Time Conversion Services

Obtaining a Delta Time in System Format

The value 10 million is expressed as 10*1000*1000 for readability. Note that the delta time value is negative.

If you use this notation, however, you are limited to the maximum number of 100-nanosecond units that can be expressed in a longword. In terms of time values, this is somewhat more than seven minutes.

9.5 Timer Requests

Timer requests made with the Set Timer (\$SETIMR) system service are queued, that is, they are ordered for processing according to their expiration times. The TQELM quota controls the number of entries a process can have pending in this timer queue.

When you call the \$SETIMR system service, you can specify either an absolute time or a delta time value. Depending on how you want the request processed, you can specify either or both of the following:

- The number of an event flag to be set when the time expires. If you do not specify an event flag, the system sets event flag 0.
- The address of an AST service routine to be executed when the time expires.

Optionally, you can specify a request identification for the timer request. You can use this identification to cancel the request, if necessary. The request identification is also passed as the AST parameter to the AST service routine, if one is specified, so that the AST service routine can identify the timer request.

The following example shows examples of timer requests using event flags and ASTs. Event flags and event flag services are described in more detail in Section 4, Event Flag Services. ASTs are described in more detail in Section 5, AST (Asynchronous System Trap) Services.

Example 1: Setting an Event Flag

```
A30SEC: .ASCID /0 00:00:30.00/ ; descriptor for ASCII 30
; seconds, delta time
B30SEC: .BLKQ 1 ; quadword to hold converted
; (binary) delta time
```

```

$BINTIM_S - ;convert to binary
TIMBUF=A30SEC, -
TIMADR=B30SEC
BSBW ERROR
❶ $SETIMR_S - ;set time to wait
EFN=#4, -
DAYTIM=B30SEC
BSBW ERROR ; call error routine
❷ $WAITFR_S - ;wait 30 seconds
EFN=#4
BSBW ERROR
```

Notes on Example 1:

- ❶ The call to \$SETIMR requests that event flag 4 be set in 30 seconds (expressed in the quadword B30SEC).

Timer and Time Conversion Services

Timer Requests

- ③ The Wait for Single Event Flag (\$WAITFR) system service places the process in a wait state until the event flag is set. When the timer expires, the flag is set and the process continues execution.

Example 2: Using an AST Service Routine

```
ANOON: .ASCID /-- 12:00:00.00/ ;descriptor for ASCII 12 noon
BNOON: .BLKQ 1 ;to hold converted (binary) noon

.
.
.
① $BINTIM_S - ;convert to binary
    TIMBUF=ANOON, -
    TIMADR=BNOON
    BSBW ERROR
② $SETIMR_S -
    DAYTIM=BNOON, - ;set timer for noon,
    ASTADR=ASTSERV, - ; specify AST routine,
    REQIDT=#12 ;request I.D. of 12 as AST parameter
    BSBW ERROR
.
.
RET

③ .ENTRY ASTSERV,"M<>" ;entry mask for AST routine
    CMPL #12,4(AP) ;is this a "noon" AST request?
    BNEQ 10$ ;if not, handle other type(s)
    ;handle "noon" AST request
.
.
RET
10$: ;handle other types of requests
.
.
RET
```

Notes on Example 2:

- ① The call to \$BINTIM converts the ASCII string representing 12:00 noon to system format. The value returned in BNOON is used as input to the \$SETIMR system service.
- ② The AST routine specified in the \$SETIMR request will be called when the timer expires, that is, at 12:00 noon. The **reqidt** argument identifies the timer request. (This argument is passed as the AST parameter and is stored at offset 4 in the argument list. See Section 5, The AST Service Routine.) The process continues execution; when the timer expires, it is interrupted by the delivery of the AST. Note that if the current time of day is past noon, the timer expires immediately.
- ③ This AST service routine checks the parameter passed by the **reqidt** argument and checks, in this example, whether it must service the 12:00 noon timer request or another type of request (identified by a different **reqidt** value). When the AST service routine completes, the process continues execution at the point of interruption.

Timer and Time Conversion Services

Timer Requests

9.5.1 Canceling Timer Requests

The Cancel Timer Request (\$CANTIM) system service cancels timer requests that have not yet been processed. The entries are removed from the timer queue. Cancellation is based on the request identification given in the timer request. For example, to cancel the request illustrated in Example 2 you would use the following call to \$CANTIM:

```
$CANTIM_S REQIDT=#12
```

If you assign the same identification to more than one timer request, all requests with that identification are canceled. If you do not specify the `reqidt` argument, all your requests are canceled.

9.6 Scheduled Wakeups

Example 1 shows a process placing itself in a wait state using the \$SETIMR and \$WAITFR services, another way for a process to make itself inactive is by hibernating. A process hibernates by issuing the Hibernate (\$HIBER) system service; hibernation is reversed by a wake-up request, which can be effected immediately with the \$WAKE system service, or scheduled with the Schedule Wakeup (\$SCHDWK) system service. For more information on the \$HIBER and \$WAKE system services, see Section 8.5.

The following example shows a process scheduling a wake-up for itself prior to hibernating.

```
ATENSEC:
    .ASCID /0 00:00:10.00/ ; descriptor for
                           ; 10-second wait time
BTENSEC:
    .BLKQ 1                ; to hold binary ten-second value

    $BINTIM_S -             ; convert time
        TIMBUF=ATENSEC, -
        TIMADR=BTENSEC
    $SCHDWK_S -             ; schedule wake
        DAYTIM=BTENSEC
    $HIBER_S                ; sleep ten seconds
```

Hibernation and wakeup are described in more detail in Section 8, Process Control Services. Note that a suitably privileged process can wake or schedule a wake-up request for another process; thus, cooperating processes can synchronize activity using hibernation and scheduled wakeups. Moreover, when you use the \$SCHDWK system service in a program, you can specify that the wake-up request be repeated at fixed time intervals.

9.6.1 Canceling Scheduled Wakeups

Scheduled wake-up requests that are pending but have not yet been processed can be canceled with the Cancel Wakeup (\$CANWAK) system service.

The following example shows the scheduling of wake-up requests for a process, CYGNUS, and the subsequent cancellation of the wakeups. The \$SCHDWK system service in this example specifies a delta time of one minute and an interval time of one minute; the wakeup is repeated every minute until the requests are canceled.

Timer and Time Conversion Services

Scheduled Wakeups

```
CYGNUS: .ASCID /CYGNUS/      ; descriptor for process name
ONE_MIN: .ASCID /0 00:01:00.00/ ; descriptor for 1 min (delta)
INTERVAL: .BLKQ 1           ; 8 bytes to hold binary 1 min
.
.
$BINTIM_S -                  ; convert to binary
    TIMBUF=ONE_MIN, -
    TIMADR=INTERVAL
.
.
$SCHDWK_S -                  ; wake up every minute
    PRCNAM=CYGNUS, -
    DAYTIM=INTERVAL, -
    REPTIM=INTERVAL
.
.
$CANWAK_S -                  ; cancel wake-ups
    PRCNAM=CYGNUS
.
.
```

9.7 Numeric and ASCII Time

The Convert Binary Time to Numeric Time (\$NUMTIM) system service converts a time in the system format into binary integer values. The service returns each of the components of the time (year, month, day, hour, and so on) into a separate word of a seven-word buffer. The \$NUMTIM system service and the format of the information returned are described in Part II.

When you need the time formatted into ASCII for inclusion in an output string, you can use the \$ASCTIM system service. The \$ASCTIM service accepts as an argument the address of a quadword that contains the time in system format and returns the date and time in ASCII format.

If you want to include the date and time in a character string that contains additional data, you can format the output string with the Formatted ASCII Output (\$FAO) system service. The \$FAO system service converts binary values to ASCII representations, and substitutes the results in character strings according to directives supplied in an input control string. Among these directives are !%T and !%D, which convert a quadword time value to an ASCII string and substitute the result in an output string. For examples of how to do this, see the discussion of \$FAO in Part II.

9.8 Setting the System Time

The Set System Time (\$SETIME) system service allows a user with the operator (OPER) and logical I/O (LOG_IO) privileges to set the current system time. You can specify a new system time (using the *timadr* argument), or you can recalibrate the current system time using the processor's hardware time-of-year clock (omitting the *timadr* argument). If you specify a time, it must be an absolute time value; a delta time (negative) value is invalid.

Timer and Time Conversion Services

Setting the System Time

The system time is set whenever the system is bootstrapped. There is normally no need to change the system time between system bootstrap operations; however, in certain circumstances you may wish to change the system time without rebooting. For example, you might specify a new system time to synchronize two processors, or to adjust for changes between standard time and daylight savings time. You might wish to recalibrate the time to ensure that the system time matches the hardware clock time (the hardware clock is more accurate than the system clock).

The \$SETIME service is called by the DCL command SET TIME.

If a process issues a delta time request and the system time is changed, the interval remaining for the request does not change; the request executes after the specified time has elapsed. If a process issues an absolute time request and the system time is changed, the request executes at the specified time, relative to the new system time.

The following example shows the effect of changing the system time on an existing timer request. In this example two set timer requests are scheduled: one is to execute after a delta time of 5 minutes, the other specifies an absolute time of 9:00.

```
.TITLE SCORPIO Show scheduled wakeups
.PSECT READ_ONLY_DATA,NOEXE,RD,NOWRT
ABS_TIME:
.ASCID /-- 9:00:00.00/          ; absolute time of 9:00 AM
DELTA_TIME:
.ASCID /0 :05:00/              ; delta time of 5 minutes
;
.PSECT WRITEABLE_DATA,NOEXE,RD,WRT
;
ABS_BINARY:
.BLKQ 1                        ; absolute time in 64-bit format
DELTA_BINARY:
.BLKQ 1                        ; delta time in 64-bit format
;
.PSECT CODE,EXE,PIC,NOSHR,RD,NOWRT
.ENTRY SCORPIO,"M<>"
$BINTIM_S -                    ; convert absolute time to
    TIMBUF=ABS_TIME, -        ; binary
    TIMADR=ABS_BINARY
BLBS RO,10$                    ; check for error
BRW ERR                        ; if so, exit
;
10$: $SETIMR_S -                ; set timer to wake AST routine
    DAYTIM=ABS_BINARY, -      ; at 9:00 AM
    ASTADR=GEMINI, -         ; routine is GEMINI
    REQIDT=#1                ; request ID number 1
BLBS RO,20$                    ; check for error
BRW ERR                        ; if so, exit
;
20$: $BINTIM_S -                ; convert delta time to
    TIMBUF=DELTA_TIME, -      ; binary
    TIMADR=DELTA_BINARY
BLBS RO,30$                    ; check for error
BRW ERR                        ; if so, exit
;
30$: $SETIMR_S -                ; set timer to wake AST routine
    DAYTIM=DELTA_BINARY, -    ; in 15 minutes
    ASTADR=GEMINI, -         ; routine is GEMINI
    REQIDT=#2                ; request ID number 2
BLBS RO,40$                    ; check for error
BRW ERR                        ; if so, exit
;
```

Timer and Time Conversion Services

Setting the System Time

```

40$: $HIBER_S ; hibernate process
;
EXIT: $EXIT_S
;
ERR: PUSHL R0
CALLS #1,G-LIB$SIGNAL
BRW EXIT
;
;
.PSECT READ_ONLY_DATA,NOEXE,RD,NOWRT
FAO_IN: .ASCID "Request ID !UB answered at !AS."
.PSECT WRITEABLE_DATA,NOEXE,RD,WRT
NOWDESC:
.LONG 12
.ADDRESS -
TIMENOW:
.BKLB 12
FAO_OUT:
.LONG 80
.ADDRESS -
FAO_STR:
.BKLB 80
;
.PSECT CODE,EXE,PIC,NOSHR,RD,NOWRT
.ENTRY GEMINI,"M<R6,R7,R8,R9,R10,R11>"
$ASCTIM_S - ; find out the current time
TIMBUF=NOWDESC, -
CVTFLG=#1 ; hours, mins, secs. only
BLBS R0,10$
BRW ERR
;
10$: $FAO_S CTRSTR=FAO_IN, - ; format string
OUTBUF=FAO_OUT, - ; place in FAO_OUT
OUTLEN=FAO_OUT, -
P1=4(AP), - ; request ID
P2=#NOWDESC ; current time
BLBS R0,20$
BRW ERR
;
20$: PUSHAL FAO_OUT
CALLS #1,G-LIB$PUT_OUTPUT
RET
;
.END SCORPIO

```

The example below shows the output received from the program shown above. Assume the program starts execution at 8:45. Seconds later, the system time is set to 9:15. The timer request that specified an absolute time of 9:00 executes immediately, because 9:00 has passed. The request that specified a delta time of 5 minutes times out at 9:20.

```

$ SHOW TIME
15-JUN-1984 8:45:04.56
$ RUN SCORPIO
-----+-----+
Request ID number 1 executed at 09:15:00.00 | operator sets system |
Request ID number 2 executed at 09:20:00.02 | time to 9:15 |
$

```

Timer and Time Conversion Services

Example Using the Timer Service

9.9 Example Using the Timer Service

```
! SYS$CREPRC options and values
INTEGER OPTIONS
EXTERNAL PROC$V_HIBER

! id of created subprocess
INTEGER CR_ID

! binary times
INTEGER TIME(2),
2     INTERVAL(2)

.

! set the PROC$V_HIBER bit in the OPTIONS mask and
! create the process
OPTIONS = IBSET (OPTIONS, %LOC(PROC$V_HIBER))
STATUS = SYS$CREPRC (CR_ID,      ! PID of created process
2     'CHECK',      ! image
2     'SLEEP',      ! process name
2     %VAL(4),      ! priority
2     ..
2     %VAL(OPTIONS)) ! hibernate
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))

! translate 6:00 a.m. (absolute time) to binary
STATUS = SYS$BINTIM ('23-- 06:00:00.00', ! 6:00 a.m.
2     TIME)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))

! translate 10 minutes (delta time) to binary
STATUS = SYS$BINTIM ('0 :10:00.00', ! 10 minutes
2     INTERVAL)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))

! schedule wake-up calls
STATUS = SYS$SCHDWK (CR_ID,      ! id of created process
2     TIME,      ! initial wake-up time
2     INTERVAL) ! repeat wake-up time
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))

.
```

To execute a program at timed intervals, you can use either LIB\$SPAWN or LIB\$CREPRC. With LIB\$SPAWN, create a subprocess that executes a command procedure containing three commands: the DCL command WAIT, the command that invokes the desired program, and a GOTO command that directs control back to the WAIT command. Since you will not want the parent process to remain in hibernation until the subprocess executes, execute the subprocess concurrently.

The following steps describe how to use SYS\$CREPRC to execute a program at timed intervals. To create a detached process, you must use SYS\$CREPRC.

- 1 Use SYS\$CREPRC to create a process that executes the desired program. Set the PROC\$V_HIBER bit of the **stsflg** argument of the SYS\$CREPRC system service to indicate that the created process should hibernate before executing the program.
- 2 Use the SYS\$SCHDWK system service to specify the time at which the system should wake the subprocess and a time interval at which the system should repeat the wake up call.

Timer and Time Conversion Services

Example Using the Timer Service

The program creates a subprocess that immediately hibernates. (The identification number of the created subprocess is returned to the parent process so that it can be passed to SYS\$SCHDWK.) The system wakes the subprocess at 6:00 a.m. the morning of the 23rd (month and year default to system month and year) and every 10 minutes thereafter.

10 Condition-Handling Services

A condition handler is a procedure that is given control when an exception occurs. An exception is an event that is detected by the hardware or software and that interrupts the execution of an image. Examples of exceptions include arithmetic overflow or underflow and reserved opcode or operand faults.

If you determine that a program needs to be informed of particular exceptions so that it can take corrective action, you can write and specify a condition handler. This condition handler, which will receive control when any exception occurs, can test for specific exceptions.

If an exception occurs and you have not specified a condition handler, the default condition handler established by the operating system is given control. If the exception is a fatal error, the default condition handler issues a descriptive message and causes the image that incurred the exception to exit.

This section describes how the condition-handling mechanism in VAX/VMS works and explains how to write a condition handler. The following system services are used in writing a condition handler.

- Set Exception Vector (\$SETEXV)
- Set System Service Failure Exception Mode (\$SETSFM)
- Unwind From Condition Handler Frame (\$UNWIND)
- Declare Change Mode or Compatibility Mode Handler (\$DCLCMH)

10.1 Types of Exception

Exceptions can be generated by any of the following:

- Hardware
- Software
- System service failures

Hardware-generated exceptions always result in conditions that require special action if program execution is to continue.

Software-generated exceptions may result in error or warning conditions. These conditions and their messages are documented in the *VAX/VMS System Messages and Recovery Procedures Reference Manual* or, for certain software routines, in the manual associated with that routine. (VAX MACRO error messages appear in the *VAX MACRO User's Guide*.)

System service failure exceptions occur when an error or severe error status is returned from a call to a system service. You can choose to handle error returns from system services by using the condition-handling mechanism rather than other error checking methods. If you want to handle exceptions generated by service failures, you must enable system service failure exception mode with the Set System Service Failure Mode (\$SETSFM) system service.

Condition-Handling Services

Types of Exception

`$SETSFM_S ENBFLG=#1`

System service failure exception mode is initially disabled, and may be enabled or disabled at any time during the execution of an image. For additional information on system service failure exception modes, see Section 2.3.4.

Table 10-1 provides a summary of common conditions caused by exceptions. In the first column the condition names are listed. The second column explains the condition more fully by giving information about the type, meaning and arguments relating to the condition. The condition type is either trap or fault. Since the explanation of types is involved you should refer to the *VAX Architecture Handbook* for more detailed information. The meaning of the exception condition is a short description of each condition. The arguments for the condition handler are listed (if any apply) which give specific information about the condition.

Table 10-1 Summary of Exception Conditions

Condition Name	Explanation						
SS\$_ACCVIO	<table><tr><td>Type:</td><td>Fault</td></tr><tr><td>Description:</td><td>Access violation</td></tr><tr><td>Arguments:</td><td><ol style="list-style-type: none">Reason for access violation. This is a mask with the format:<ul style="list-style-type: none">Bit 0 = type of access violation<ul style="list-style-type: none">0 = page table entry protection code did not permit intended access1 = POLR, P1LR, or SLR length violationBit 1 = page table entry reference<ul style="list-style-type: none">0 = specified virtual address not accessible1 = associated page table entry not accessibleBit 2 = intended access<ul style="list-style-type: none">0 = read1 = modifyVirtual address to which access was attempted</td></tr></table>	Type:	Fault	Description:	Access violation	Arguments:	<ol style="list-style-type: none">Reason for access violation. This is a mask with the format:<ul style="list-style-type: none">Bit 0 = type of access violation<ul style="list-style-type: none">0 = page table entry protection code did not permit intended access1 = POLR, P1LR, or SLR length violationBit 1 = page table entry reference<ul style="list-style-type: none">0 = specified virtual address not accessible1 = associated page table entry not accessibleBit 2 = intended access<ul style="list-style-type: none">0 = read1 = modifyVirtual address to which access was attempted
Type:	Fault						
Description:	Access violation						
Arguments:	<ol style="list-style-type: none">Reason for access violation. This is a mask with the format:<ul style="list-style-type: none">Bit 0 = type of access violation<ul style="list-style-type: none">0 = page table entry protection code did not permit intended access1 = POLR, P1LR, or SLR length violationBit 1 = page table entry reference<ul style="list-style-type: none">0 = specified virtual address not accessible1 = associated page table entry not accessibleBit 2 = intended access<ul style="list-style-type: none">0 = read1 = modifyVirtual address to which access was attempted						
SS\$_ARTRES	<table><tr><td>Type:</td><td>Trap</td></tr><tr><td>Description:</td><td>Reserved arithmetic trap</td></tr><tr><td>Argument:</td><td>None</td></tr></table>	Type:	Trap	Description:	Reserved arithmetic trap	Argument:	None
Type:	Trap						
Description:	Reserved arithmetic trap						
Argument:	None						

Condition-Handling Services

Types of Exception

Table 10-1 (Cont.) Summary of Exception Conditions

Condition Name	Explanation	
SS\$_ASTFLT	Type:	Trap
	Description:	Stack invalid during attempt to deliver an AST
	Arguments:	<ol style="list-style-type: none"> 1 Stack pointer value when fault occurred 2 AST parameter of failed AST 3 Program counter (PC) at AST delivery interrupt 4 Processor status longword (PSL) at AST delivery interrupt¹ 5 Program counter (PC) to which AST would have been delivered¹ 6 Processor status longword (PSL) to which AST would have been delivered¹
SS\$_BREAK	Type:	Fault
	Description:	Breakpoint instruction encountered
	Arguments:	None
SS\$_CMODSUPR	Type:	Trap
	Description:	Change mode to supervisor instruction encountered ²
	Arguments:	Change mode code. The possible values are -32,768 through 32,767.
SS\$_CMODUSER	Type:	Trap
	Description:	Change mode to user instruction encountered ²
	Arguments:	Change mode code. The possible values are -32,768 through 32,767.

¹The PC and PSL normally included in the signal array are not included in this argument list. The stack pointer of the access mode receiving this exception is reset to its initial value.

²If a change mode handler has been declared for user or supervisor modes with the Declare Change Mode or Compatibility Mode Handler (\$DCLCMH) system service, that routine receives control when the associated trap occurs.

Condition-Handling Services

Types of Exception

Table 10-1 (Cont.) Summary of Exception Conditions

Condition Name	Explanation	
SS\$_COMPAT	Type:	Fault
	Description:	Compatibility mode exception. This exception condition can only occur when executing in compatibility mode. ³
	Arguments:	Type of compatibility exception. The possible values are: <ul style="list-style-type: none">• 0 = Reserved instruction execution• 1 = BPT instruction executed• 2 = IOT instruction executed• 3 = EMT instruction executed• 4 = TRAP instruction executed• 5 = Illegal instruction executed• 6 = Odd address fault• 7 = TBIT trap
SS\$_DECOVF	Type:	Trap
	Description:	Decimal overflow
	Arguments:	None
SS\$_FLT DIV	Type:	Trap
	Description:	Floating/decimal divide by zero
	Arguments:	None
SS\$_FLT DIV_F	Type:	Fault
	Description:	Floating divide by zero fault
	Arguments:	None
SS\$_FLT OVF	Type:	Trap
	Description:	Floating overflow
	Arguments:	None
SS\$_FLT OVF_F	Type:	Fault
	Description:	Floating overflow fault
	Arguments:	None

³If a compatibility mode handler has been declared with the Declare Change Mode or Compatibility Mode Handler (\$DCLCMH) system service, that routine receives control when this fault occurs.

Condition-Handling Services

Types of Exception

Table 10-1 (Cont.) Summary of Exception Conditions

Condition Name	Explanation	
SS\$_FLTUND	Type:	Trap
	Description:	Floating underflow
	Arguments:	None
SS\$_FLTUND_F	Type:	Fault
	Description:	Floating underflow fault
	Arguments:	None
SS\$_INTDIV	Type:	Trap
	Description:	Integer divide by zero
	Arguments:	None
SS\$_INTOVF	Type:	Trap
	Description:	Integer overflow
	Arguments:	None
SS\$_OPCCUS	Type:	Fault
	Description:	Opcode reserved to customer fault
	Arguments:	None
SS\$_OPCDEC	Type:	Fault
	Description:	Opcode reserved to DIGITAL fault
	Arguments:	None
SS\$_PAGRDERR	Type:	Fault
	Description:	Read error occurred during an attempt to read a faulted page from disk
	Arguments:	Translation not valid reason. This is a mask with the format: <ul style="list-style-type: none">• Bit 0 = 0• Bit 1 = page table entry reference<ul style="list-style-type: none">• 0 = specified virtual address not valid• 1 = associated page table entry not valid• Bit 2 = intended access<ul style="list-style-type: none">• 0 = read• 1 = modify

Condition-Handling Services

Types of Exception

Table 10–1 (Cont.) Summary of Exception Conditions

Condition Name	Explanation	
SS\$_RADRMOD	Type:	Fault
	Description:	Attempt to use a reserved addressing mode
	Arguments:	None
SS\$_ROPRAND	Type:	Fault
	Description:	Attempt to use a reserved operand
	Arguments:	None
SS\$_SSFAIL	Type:	Fault
	Description:	System service failure (when system service failure exception mode is enabled)
	Arguments:	Status return from system service (R0) (The same value is in R0 of the mechanism array)
SS\$_SUBRNG	Type:	Trap
	Description:	Subscript range trap
	Arguments:	None
SS\$_TBIT	Type:	Fault
	Description:	Trace bit is pending following an instruction
	Arguments:	None

10.1.1 Change Mode and Compatibility Mode Handlers

Two types of hardware exception can be handled in a special way, bypassing the normal condition-handling mechanism described in this section.

- Traps caused by change mode to user or change mode to supervisor instructions
- Compatibility mode faults

You can use the Declare Change Mode or Compatibility Mode Handler (\$DCLCMH) system service to establish procedures to receive control when one of these conditions occurs. The \$DCLCMH system service is described in Part II.

10.2 How to Specify Condition Handlers

You can establish condition handlers to receive control in the event of an exception in two ways.

- 1 By specifying the address of the entry mask of a condition handler in the first longword of a procedure call frame
- 2 By establishing exception handlers with the Set Exception Vector (\$SETEXV) system service

The first of these methods is the preferred way to specify a condition handler for a particular image. The use of call frame handlers is also the most efficient way in terms of declaration. Vector handlers should be used for special purposes, such as writing debuggers. The VAX MACRO programmer can use a single move address instruction to place the address of the condition handler in the longword pointed to by the current frame pointer (FP).

```
MOVAB    HANDLER, (FP)
```

The high-level language programmer can call the common Run-Time Library routine LIB\$ESTABLISH (see the *VAX/VMS Run-Time Library Routines Reference Manual*); however, some languages provide access to condition handling as part of the language.

Each procedure on the call stack can declare a condition handler.

The \$SETEXV system service allows you to specify addresses for a primary exception handler, a secondary exception handler, and a last chance exception handler. Handlers may be specified for each access mode. The primary exception vector is reserved for the debugger. In general, use of the vectored handlers should be avoided unless absolutely necessary. If you use a vectored handler, note that it must be prepared for all exceptions occurring in that access mode.

An address of 0 in the first longword of a procedure call frame or in an exception vector indicates that no condition handler exists for that call frame or vector.

10.3 The Exception Dispatcher

When an exception occurs, control is passed to the operating system's exception dispatching routine. The exception dispatcher searches for a condition-handling routine using the following search order:

- 1 The primary exception vector for the access mode at which the program was executing when the exception occurred.
- 2 The secondary exception vector for the access mode at which the program was executing when the exception occurred.
- 3 The condition handler address specified in the procedure call stack of the access mode at which the program was executing when the exception occurred. Call frames on the stack are scanned backwards, using the saved frame pointer in each call frame to refer to the previous call frame.
- 4 The last chance exception vector for the access mode at which the program was executing when the exception occurred.

Condition-Handling Services

The Exception Dispatcher

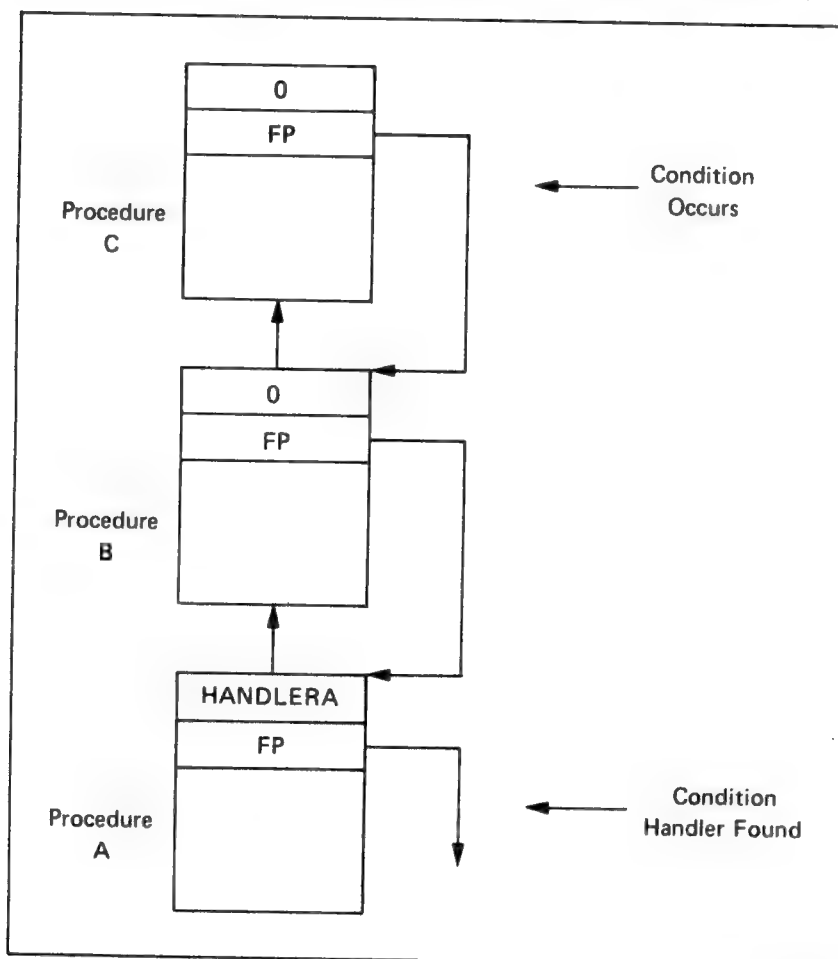
The search is terminated when the dispatcher finds a condition handler. If the dispatcher cannot find a user-specified condition handler, it calls the condition handler whose address is stored in the last chance exception vector. If the image was activated by the command interpreter, the last chance vector points to the catch-all condition handler. The catch-all handler issues a message and either continues program execution or causes the image to exit, depending on whether the condition was a warning or an error condition, respectively.

The catch-all handler may be called in two ways.

- 1 If the last chance exception vector returns to the dispatcher, or if the last chance exception vector is empty, it calls the catch-all condition handler, and exits with the return status code `SS$_NOHANDLER`.
- 2 If the exception dispatcher detects an access violation, it calls the catch-all condition handler, and exits with the return status code `SS$_ACCVIO`.

Figure 10-1 illustrates the exception dispatcher's search of the call stack for a condition handler.

Figure 10-1 Search of Stack for Condition Handler



ZK-858-82

Condition-Handling Services

The Exception Dispatcher

Notes on Figure 10-1:

- 1 The illustration of the call stack indicates the calling sequence: Procedure A called Procedure B, and Procedure B called Procedure C. Procedure A established a condition handler.
- 2 An exception occurs while Procedure C is executing. The exception dispatcher searches for a condition handler.
- 3 After checking for a condition handler declared in the exception vectors (assume that none has been specified for this process), the dispatcher looks at the first longword of Procedure C's call frame. A value of 0 indicates that no condition handler has been specified. The dispatcher locates the call frame for Procedure B by using the frame pointer (FP) in Procedure C's call frame. Again, it finds no condition handler, and locates Procedure A's call frame.
- 4 The dispatcher locates and gives control to HANDLER A.

10.4 The Argument List Passed to a Condition Handler

When the dispatcher finds a condition handler, it passes control to it using a CALLG instruction. The argument list passed to the condition handler is constructed on the stack and consists of the addresses of two argument arrays, as illustrated in Figure 10-2; these arguments are described in detail in Sections 10.4.1 and 10.4.2.

You can define the following symbolic names to refer to these arguments using the \$CHFDEF macro instruction.

Symbolic Offset	Value
CHF\$_SIGARGLST	Address of signal array
CHF\$_MCHARGLST	Address of mechanism array
CHF\$_SIG_ARGS	Number of signal arguments
CHF\$_SIG_NAME	Condition name
CHF\$_SIG_ARG1	First signal-specific argument
CHF\$_MCH_ARGS	Number of mechanism arguments
CHF\$_MCH_FRAME	Establisher frame address
CHF\$_MCH_DEPTH	Frame depth of establisher
CHF\$_MCH_SAVRO	Saved register 0
CHF\$_MCH_SAVR1	Saved register 1

10.4.1 Signal Array Arguments

The signal array contains the following values describing the condition.

- 1 Condition name—The symbolic value assigned to the specific condition. The possible exception conditions and their symbolic definitions are listed in Table 10-1.
- 2 Arguments—Specific information relating to the condition. Table 10-1 also shows the arguments provided with each condition.

Condition-Handling Services

The Argument List Passed to a Condition Handler

- 3 PC—The program counter at the time of the exception. Depending on the type of exception (fault or trap), this can be the address of the instruction that caused the exception (for a fault), or of the following instruction (for a trap).
- 4 PSL—The processor status longword at the time of the exception.

10.4.2 Mechanism Array Arguments

The mechanism array describes the context in which the exception occurred. The following arguments are supplied.

- 1 Establisher frame—The frame pointer (FP) registers contents of the call frame that established the condition handler. This is the address of the longword containing the condition handler address. For example, if the call stack is as shown earlier in Figure 10-1, this argument points to the call frame for Procedure A.

This value can be used to display local variables in the procedure that established the condition handler, if the variables are at known offsets from the FP of the procedure.

- 2 Depth—The frame number of the procedure that established the condition handler, relative to the frame of the procedure that incurred the exception. The depth is determined as described below.

Depth	Meaning
-3	Condition handler was established in the last chance exception vector
-2	Condition handler was established in the primary exception vector
-1	Condition handler was established in the secondary exception vector
0	Condition handler was established by the frame that was active when the exception occurred
1	Condition handler was established by the caller of the frame that was active when the exception occurred
2	Condition handler was established by the caller of the caller of the frame that was active when the exception occurred
.	and so on
.	
.	

For example, if the call stack is as shown earlier in Figure 10-1, the depth argument passed to HANDLER would have a value of 2.

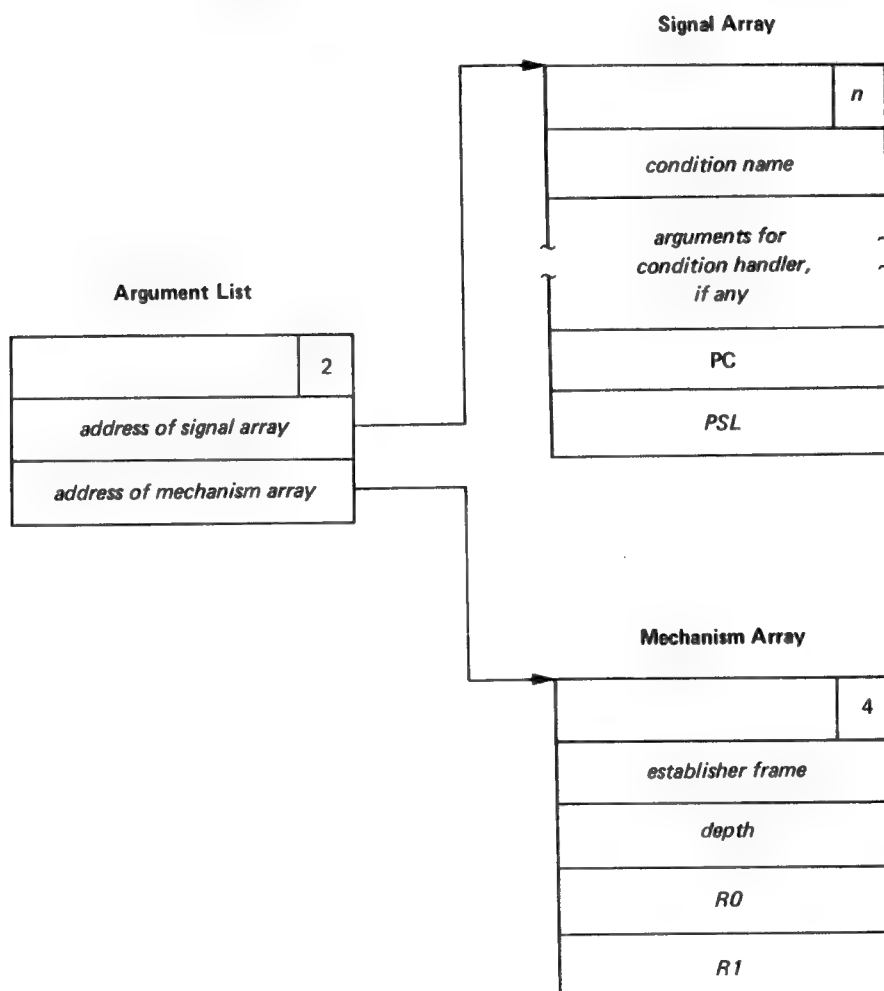
The condition handler can use this argument to determine whether it wants to handle the condition. For example, the handler may not want to handle the condition if the exception that caused the condition did not occur in the establisher frame.

- 3 R0—The contents of register 0 when the exception occurred.
- 4 R1—The contents of register 1 when the exception occurred.

Condition-Handling Services

Courses of Action for the Condition Handler

Figure 10-2 Argument List and Arrays Passed to Condition Handler



You can define symbolic names to refer to these arguments using the **\$CHFDEF** macro instruction. The symbolic names are:

Symbolic Offset	Value
CHF\$_SIGARGLST	Address of signal array
CHF\$_MCHARGLST	Address of mechanism array
CHF\$_SIG_ARGS	Number of signal arguments
CHF\$_SIG_NAME	Condition name
CHF\$_SIG_ARG1	First signal-specific argument
CHF\$_MCH_ARGS	Number of mechanism arguments
CHF\$_MCH_FRAME	Establisher frame address
CHF\$_MCH_DEPTH	Frame depth of establisher
CHF\$_MCH_SAVR0	Saved register 0
CHF\$_MCH_SAVR1	Saved register 1

ZK-859-82

Condition-Handling Services

Courses of Action for the Condition Handler

10.5 Courses of Action for the Condition Handler

After the condition-handling routine determines the nature of the exception, it can take one of the three following courses of action.

1 Continue

The condition handler may or may not be able to fix the problem, but the program can attempt to continue execution. The handler places the return status value `SS$_CONTINUE` in `R0` and issues a `RET` instruction to return control to the dispatcher. If the exception was a fault, the instruction that caused it is reexecuted; if the exception was a trap, control is returned at the instruction following the one that caused it.

2 Resignal

The handler cannot fix the problem, or this condition is one that it does not handle. It places the return status value `SS$_RESIGNAL` in `R0` and issues a `RET` instruction to return control to the exception dispatcher. The dispatcher resumes its search for a condition handler. If it finds another condition handler, it passes control to that routine.

3 Unwind

The condition handler cannot fix the problem, and execution cannot continue using the current flow. The handler issues the Unwind Call Stack (`$UNWIND`) system service to unwind the call stack. Call frames may then be removed from the stack and the flow of execution modified, depending on the arguments to the `$UNWIND` service.

Examples of these three situations are shown in the next two sections.

10.5.1 Example of Condition-Handling Routines

The following example shows two procedures, A and B, that have declared condition handlers. The notes describe the sequence of events that would occur if a call to a system service failed during the execution of Procedure B.

```
.ENTRY PGMA,"M<>" ; entry mask for
                        ; procedure A
1 MOVAB HANDLER, (FP) ; declare condition handler
  $SETSPM_S -
      ENBFLG=#1      ; enable SSFAIL
                        ; exceptions
2 CALLG  ARGLIST,PGMB ; call procedure B
.
3 .ENTRY HANDLER,"M<R2,R3,R4>" ; entry mask of HANDLER
  MOVL   CHF$L_SIGARGLST(AP),R4 ; get addr of signal args
  CMPL   $$$_SSFAIL,CHF$L_SIG_NAME(R4)
      BNEQ 10$ ; system service failure?
                        ; no - resignal
                        ; handle SSFAIL exception
4
  MOVZWL $$$_CONTINUE,R0 ; signal to continue
  RET ; return to exception
                        ; dispatcher
10$: MOVZWL $$$_RESIGNAL,R0 ; signal to resignal
  RET ; return to dispatcher
.ENTRY PGMB,"M<R2,R3,R4>" ; entry mask of procedure B
```

Condition-Handling Services

Courses of Action for the Condition Handler

[illegible]

Notes on Example:

- ① Procedure A executes and establishes condition handler HANDLERA. HANDLERA is set up to respond to exceptions caused by failures in system service calls.
- ② During its execution, Procedure A calls Procedure B.
- ③ Procedure B establishes condition handler HANDLERB. HANDLERB is set up to respond to breakpoint faults.
- ④ While Procedure B is executing, an exception occurs caused by a system service failure.
- ⑤ The exception dispatcher searches the exception vectors for a condition handler (assume there are none defined), and then searches the call stack. HANDLERB is called with the condition SS\$_SSFAIL.
- ⑥ Since HANDLERB only handles breakpoint faults, it places the return value SS\$_RESIGNAL in R0 and returns control to the exception dispatcher.
- ⑦ The exception dispatcher resumes its search for a condition handler and calls HANDLERA.
- ⑧ HANDLERA handles the system service failure exception, corrects the condition, places the return value SS\$_CONTINUE in R0, and returns control to the exception dispatcher.
- ⑨ The dispatcher returns control to Procedure B, and execution of Procedure B resumes at the instruction following the system service failure.

10.5.2 Unwinding the Call Stack

The third course of action a condition handler can take is to unwind the procedure call stack. The unwind operation is complex, and should be used only when control must be restored to an earlier procedure in the calling sequence. Moreover, use of the \$UNWIND system service requires the calling condition handler to be aware of the calling sequence and of the exact point to which control is to return.

Condition-Handling Services

Courses of Action for the Condition Handler

The \$UNWIND system service accepts two optional arguments.

- 1 The depth to which the unwind is to occur. If the depth is 1, the call stack is unwound to the caller of the procedure that incurred the exception. If the depth is 2, the call stack is unwound to the caller's caller, and so on. By specifying the depth in the mechanism array, the handler can unwind to the procedure that established the handler.
- 2 The address of a location to receive control when the unwind operation is complete, that is, a PC to replace the current PC in the call frame of the procedure that will receive control when all specified frames have been removed from the stack.

If no argument is supplied to the \$UNWIND service, the unwind is performed to the caller of the procedure that established the condition handler that is issuing the \$UNWIND service. Control is returned to the address specified in the return PC for that procedure. Note that this is the default and normal case for unwinding.

Another common case of unwinding is to unwind to the procedure that declared the handler. This is easily done by using the depth value from the exception mechanism array (CHF\$L_MCH_DEPTH) as the depth argument to \$UNWIND.

It therefore follows that the default unwind (no depth specified) is equivalent to specifying CHF\$L_MCH_DEPTH plus one. However, in certain cases of nested exceptions this is not the case. DIGITAL recommends that you omit the depth argument when unwinding to the caller of the routine that established the condition handler.

Figure 10-3 illustrates an unwind situation and describes some of the possible results.

The unwind operation consists of two parts.

- 1 In the call to \$UNWIND, the return PC's saved in the stack are modified to point into a routine within the \$UNWIND service, but the entire stack remains present.
- 2 When the handler returns, control is directed to this routine by the modified PC's. It proceeds to return to itself, removing the modified stack frames, until the stack has been unwound to the proper depth.

For this reason, the stack is in an intermediate state directly after calling \$UNWIND. Handlers should in general return immediately after calling \$UNWIND.

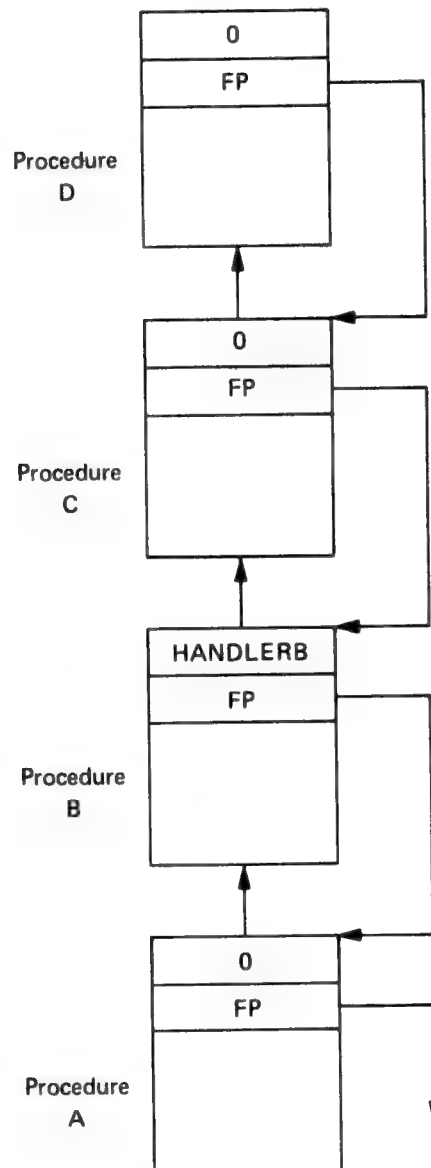
During the actual unwinding of the call stack, the unwind routine examines each frame in the call stack to see if a condition handler has been declared. If a handler has been declared, the unwind routine calls the handler with the status value SS\$_UNWIND (indicating that the call stack is being unwound) in the condition name argument of the signal array. When a condition handler is called with this status value, it can perform any procedure-specific clean-up operations required. After the handler returns, the call frame is removed from the stack.

Thus, in Figure 10-3, HANDLERB may be called a second time, during the unwind operation. Note that HANDLERB does not have to be able to specifically interpret the SS\$_UNWIND status value; the RET instruction merely returns control to the unwind procedure, which does not check any status values.

Condition-Handling Services

Courses of Action for the Condition Handler

Figure 10-3 Unwinding the Call Stack



ZK-860-82

Notes on Figure 10-3:

- 1 The procedure call stack is as shown. Assume that no exception vectors are declared for the process and that the exception occurs during the execution of Procedure D.
- 2 Since neither Procedure D nor Procedure C has established a condition handler, HANDLERB receives control.
- 3 If HANDLERB issues the \$UNWIND system service with no arguments, the call frames for B, C, and D are removed from the stack (along with the call frame for HANDLERB itself), and control returns to Procedure A. Procedure A receives control at the point following its call to Procedure B.

Condition-Handling Services

Courses of Action for the Condition Handler

- 4 If HANDLERB issues the \$UNWIND system service specifying a depth of 2, call frames for C and D are removed, and control returns to Procedure B.

10.6 Multiple Exceptions

It is possible for a second exception to occur while a condition handler or a procedure that it has called is still executing. In this case, when the exception dispatcher searches for a condition handler, it skips the frames that were searched to locate the first handler.

The search for a second handler terminates in the same manner as the initial search, as described in Section 10.3.

If the \$UNWIND system service is issued by the second active condition handler, the depth of the unwind is determined according to the same rules followed in the exception dispatcher's search of the stack: all frames that were searched for the first condition handler are skipped.

Primary and secondary vectored handlers, on the other hand, are always entered when an exception occurs.

If an exception occurs during the execution of a handler established in the primary or secondary exception vector, that handler must handle the additional condition. Failure to do so correctly may result in a recursive exception loop in which the vectored handler is repeatedly called until the user stack is exhausted.

10.7 Example Using Condition-Handling Services

```
! arguments for exit handler
INTEGER*4 EXIT_STATUS      ! status
INTEGER*4 FLAG /64/
! Setup for exit handler
STRUCTURE /EXIT_DESCRIPTOR/
  INTEGER LINK,
  2      ADDR,
  2      ARGS /2/,
  2      STATUS_ADDR,
  2      FLAG_ADDR
END STRUCTURE
RECORD /EXIT_DESCRIPTOR/ HANDLER
! exit handler
EXTERNAL EXIT_HANDLER
INTEGER*4 STATUS,
2      SYS$ASCEFC,
2      SYS$SETEF
! associate with the common event flag
! cluster and set the flag
STATUS = SYS$ASCEFC (XVAL(FLAG),
2      'ALIVE',,)
IF (.NOT. STATUS) CALL LIB$SIGNAL (XVAL(STATUS))
STATUS = SYS$SETEF (XVAL(FLAG))
IF (.NOT. STATUS) CALL LIB$SIGNAL (XVAL (STATUS))
! Do not exit until cooperating program has a chance to
! associate with the common event flag cluster.
! Enter the handler and argument addresses
! into the exit handler description
HANDLER.ADDR = %LOC(EXIT_HANDLER)
HANDLER.STATUS_ADDR = %LOC(EXIT_STATUS)
HANDLER.FLAG_ADDR = %LOC(FLAG)
```

Condition-Handling Services

Example Using Condition-Handling Services

```
! Establish the exit handler
CALL SYS$DCLEXH (HANDLER)
! continue with program

END

! Exit Subroutine
SUBROUTINE CLEAR_FLAG (EXIT_STATUS,
2      FLAG)
! Exit handler clears the event flag
! Declare dummy argument
INTEGER EXIT_STATUS,
2      FLAG
! declare status variable and system routine
INTEGER STATUS,
2      SYS$ASCEFC,
2      SYS$CLREF
! associate with the common event flag
! cluster and clear the flag
STATUS = SYS$ASCEFC (XVAL(FLAG),
2      'ALIVE',)
IF (.NOT. STATUS) CALL LIB$SIGNAL (XVAL(STATUS))
STATUS = SYS$CLREF (XVAL(FLAG))
IF (.NOT. STATUS) CALL LIB$SIGNAL (XVAL (STATUS))
```

An exit handler should be written as a subroutine since no function value can be returned. The dummy arguments of the exit subroutine should agree in number, order, and data type with the arguments you specified in the call to SYS\$DCLEXH.

Assume that two or more programs are cooperating with each other. To keep track of which programs are executing, each has been assigned a common event flag (the common event flag cluster is named ALIVE). When a program begins, it sets its flag; when the program terminates it clears its flag. Since it is important that each program clear its flag before exiting, you create an exit handler (as in the example above) to perform the action. The exit handler accepts two arguments, the final status of the program and the number of the event flag to be cleared.

Since, in this example, the clean-up operation is to be performed regardless of whether the program completed successfully, the final status is not examined in the exit routine.

11 Memory Management Services

The VAX/VMS memory management routines map and control the relationship between physical memory and a process's virtual address space. These activities are, for the most part, transparent to you as a user and to your programs. However, you can in some cases make a program more efficient by explicitly controlling its virtual memory usage. The following system services are memory management services.

- Expand Program/Control Region (\$EXPREG)
- Contract Program/Control Region (\$CNTREG)
- Create Virtual Address Space (\$CRETVA)
- Delete Virtual Address Space (\$DELTVA)
- Create and Map Section (\$CRMPSC)
- Map Global Section (\$MGBLSC)
- Delete Global Section (\$DGBLSC)
- Update Section File on Disk (\$UPDSEC)
- Lock Pages in Working Set (\$LKWSET)
- Unlock Pages from Working Set (\$ULWSET)
- Adjust Working Set Limit (\$ADJWSL)
- Purge Working Set (\$PURGWS)
- Lock Page in Memory (\$LCKPAG)
- Unlock Page in Memory (\$ULKPAG)
- Set Protection on Pages (\$SETPRT)
- Set Process Swap Mode (\$SETSWM)
- Set Stack Limits (\$SETSTK)

Memory management services allow you to control the size of virtual and physical memory address space available to a program. For example, they allow you to do the following:

- Increase or decrease the virtual address space available in a process's program or control region
- Control the process's working set size and the exchange of pages between physical memory and the paging device
- Define disk files containing data or shareable images and map the files into the process's virtual address space

Memory Management Services

This section discusses the services that provide these capabilities. However, before you use any of these services, you should have an understanding of the VAX memory structure and memory management routines. Where pertinent, virtual memory concepts related to the use of particular services are discussed in this section. For more background information, see the *VAX/VMS Summary Description and Glossary*.

11.1 Virtual Address Space

The virtual address space of a process is divided into two regions:

- The program region (P0), which contains the image currently being executed.
- The control region (P1), which contains the information maintained by the system on behalf of the process. It also contains the user stack, which expands toward the lower-addressed end of the control region.

Figure 11-1 illustrates the layout of a process's virtual memory. The initial size of a process's virtual address space depends on the size of the image being executed.

To facilitate memory protection and mapping, the virtual address space is subdivided into 512-byte units called pages. Using memory management services, a process can add a specified number of pages to the end of either the program region or the control region. Adding pages to the program region provides the process with additional space for image execution, for example, for the dynamic creation of tables or data areas. Adding pages to the control region increases the size of the user stack. As new pages are referenced the stack is automatically expanded. (The user stack can also be expanded when the image is linked, by the use of the `STACK=` option in a linker options file.)

The maximum size to which a process can increase its address space is controlled by the `SYSGEN` parameter `VIRTUALPAGECNT`.

11.2 Increasing and Decreasing Virtual Address Space

The Expand Program/Control Region (`$EXPREG`) system service adds pages to the end of either the program or control region, and optionally returns the range of virtual addresses of the new pages. For example, if you want to add four pages to a process's program region, you can write a call to the `$EXPREG` system service as follows:

```
BEGSPACE:
    .BLKL    2                ; 2 longwords to hold start
                                ; and end of new pages

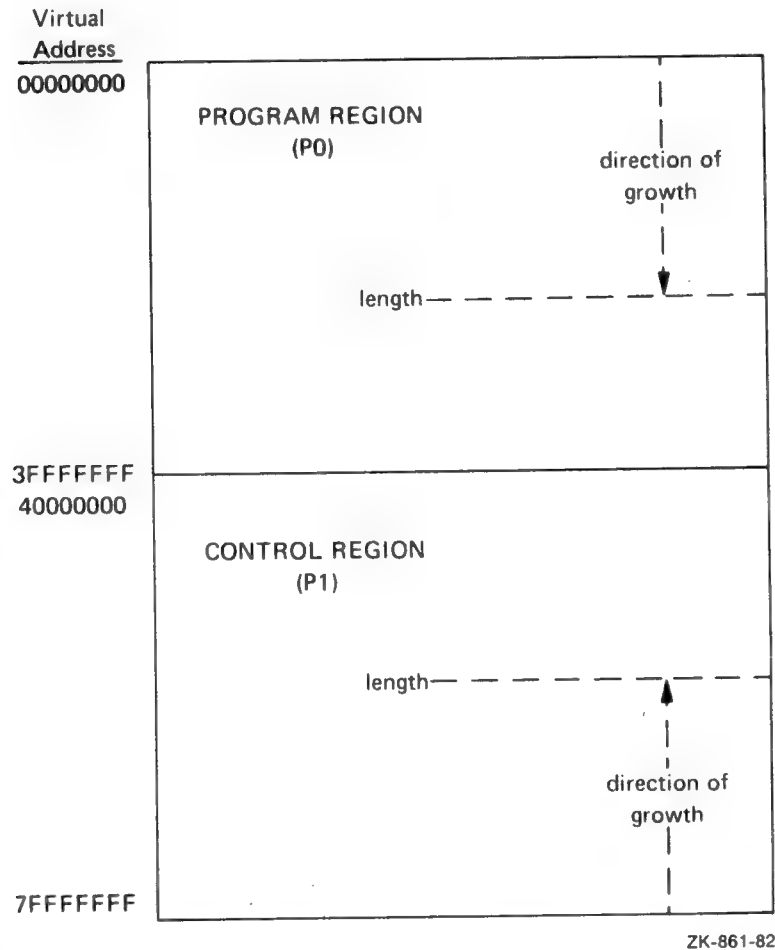
    $EXPREG_S -                ; get 4 pages
    PAGCNT=#4, -
    RETADR=BEGSPACE, -
    REGION=#0
```

The value 0 is passed in the `region` argument to specify that the pages are to be added to the program region. To add the same number of pages to the control region, you would specify `REGION=#1`.

Memory Management Services

Increasing and Decreasing Virtual Address Space

Figure 11-1 Layout of Process Virtual Address Space



Note that the **region** argument to the \$EXPREG service is optional; if not specified, the pages are added to or deleted from the program region by default.

The \$EXPREG service can only add pages at the end of a particular region. When you need to add pages that are not at the end of these regions, you can use the Create Virtual Address Space (\$CRETVA) system service. Likewise, when you need to delete pages created by either \$EXPREG or \$CRETVA, you can use the Delete Virtual Address Space (\$DELTVA) system service. For example, if you have used the \$EXPREG service twice to add pages to the program region, and want to delete the first range of pages but not the second, you could use the \$DELTVA system service as shown in the following example.

Memory Management Services

Increasing and Decreasing Virtual Address Space

```
BEGSPACEA:
    .BLKL  2          ; start and end of 1st area
BEGSPACEB:
    .BLKL  2          ; start and end of 2nd area

    .
    .
$EXPREG_S -          ; four pages
    PAGCNT=#4, -
    RETADR=BEGSPACEA, -
    REGION=#0
BSBW  ERROR

    .
    .
$EXPREG_S -          ; three more
    PAGCNT=#3, -
    RETADR=BEGSPACEB, -
    REGION=#0
BSBW  ERROR

    .
    .
$DELTVA_S -          ; delete first 4 pages
    INADR=BEGSPACEA
BSBW  ERROR
```

In the above example, the first call to \$EXPREG adds four pages to the program region; the virtual addresses of the created pages are returned in the two-longword array at BEGSPACEA. The second call adds three pages, and returns the addresses at BEGSPACEB. The call to \$DELTVA deletes the first four pages that were added.

Note: Do not use the \$CNTREG, or \$CRETVA system services in conjunction with other user-written procedures and/or DIGITAL-supplied procedures (including Run-Time Library procedures). These system services provide no means to communicate a change in virtual address space with other routines. DIGITAL recommends that you use either \$EXPREG or the Run-time Library Procedures Allocate Virtual Memory (LIB\$GET_VM) to get memory (whichever is appropriate). Documentation on this routine is in the *VAX/VMS Run-Time Library Routines Reference Manual*. When using \$DELTVA, the user should take care to only delete pages that he specifically created.

11.2.1 Input Address Arrays and Return Address Arrays

When the \$EXPREG system service adds pages to a region, it adds them in the normal direction of growth for the region. The return address array, if requested, indicates the order in which the pages were added.

- If the program region is expanded, the starting virtual address is smaller than the ending virtual address.
- If the control region is expanded, the starting virtual address is larger than the ending virtual address.

The direction of contraction produced by the \$CNTREG system service is from a higher to a lower address in the program region and from a lower to a higher address in the control region.

The addresses returned indicate the first byte in the first page that was added or deleted and the last byte in the last page that was added or deleted.

Memory Management Services

Increasing and Decreasing Virtual Address Space

When input address arrays are specified for the Create or Delete Virtual Address Space system services (\$CRETVA and \$DELTVA, respectively), these services add or delete pages beginning with the address specified in the first longword and ending with the address specified in the second longword.

The order in which the pages are added or deleted does not have to be in the normal direction of growth for the region. Moreover, since these services add or delete only whole pages, they ignore the low-order 9 bits of the specified virtual address (the low-order 9 bits contain the byte offset within the page). The virtual addresses returned indicate the byte offsets.

Table 11-1 shows some sample virtual addresses that might be specified as input to \$CRETVA or \$DELTVA and shows the return address arrays, if all pages are successfully added or deleted.

Table 11-1 Sample Virtual Address Arrays

Input Array		Region	Output Array		Number of Pages
Start	End		Start	End	
1010	1670	P0	1000	17FF	4
1450	1451	P0	1400	15FF	1
1200	1000	P0	1000	13FF	2
1450	1450	P0	1400	15FF	1
7FFEC010	7FFEC010	P1	7FFEC1FF	7FFEC000	1
7FFEC010	7FFEBCA0	P1	7FFEC1FF	7FFEB000	3

Note that if the input virtual addresses are the same, as in the fourth and fifth items in Table 11-1, a single page is added or deleted. The return address array indicates that the page was added or deleted in the normal direction of growth for the region.

11.3 Page Ownership and Page Protection

Each page in a process's virtual address space is owned by the access mode that created the page. For example, pages in the program region initially provided for the execution of an image are owned by user mode. Pages that the image creates dynamically are also owned by user mode. Pages in the control region, except for the pages containing the user stack, are normally owned by more privileged access modes.

Only the owner access mode or a more privileged access mode can delete the page or otherwise affect it. The owner of a page can also indicate, by means of a protection code, the type of access that each access mode will be allowed.

The Set Protection on Pages (\$SETPRT) system service changes the protection assigned to a page or group of pages. The protection is expressed as a code that indicates the specific type of access (none, read-only, or read/write) for each of the four access modes (kernel, executive, supervisor, user). Only the owner access mode or a more privileged access mode can change the protection for a page.

Memory Management Services

Page Ownership and Page Protection

When an image attempts to access a page that is protected against the access attempted, a hardware exception called an access violation occurs. When an image calls a system service, the service probes the pages to be used to determine whether an access violation would occur when the image attempted to read or write one of the pages. If an access violation would occur, the service exits with the status code `SS$_ACCVIO`.

Since the memory management services add, delete, or modify a single page at a time, one or more pages can be successfully changed before an access violation is detected. If the `retadr` argument is specified in the service call, the service returns the addresses of pages changed (added, deleted, or modified) before the error. If no pages are affected, that is, if an access violation would occur on the first page specified, the service returns a -1 in both longwords of the return address array.

If the `retadr` argument is not specified, no information is returned.

11.4 Working Set Paging

When a process is executing an image, a subset of its pages resides in physical memory; these pages are called the process's working set. The working set includes pages in both the program region and the control region.

When the image refers to a page that is not in memory, a page fault occurs, and the page is brought into memory, replacing an existing page in the working set. If the page that is going to be replaced has been modified during the execution of the image, that page is written into a paging file on disk. When this page is needed again, it is brought back into memory, again replacing a current page from the working set. This exchange of pages between physical memory and secondary storage is called paging.

The paging of a process's working set is transparent to the process. However, if a program is very large, or if pages in the program image that are heavily used are being paged in and out frequently, the overhead required for paging may decrease the program's efficiency. Some system services allow a process, within limits, to counteract these potential problems.

- The Adjust Working Set Limit (`$ADJWSL`) system service increases or decreases the maximum number of pages that a process can have in its working set.
- The Purge Working Set (`$PURGWS`) system service removes one or more pages from the working set.
- The Lock Pages in Working Set (`$LKWSET`) system service makes one or more pages in the working set ineligible for paging.

The initial size of a process's working set is defined by the process's working set default (`WSDEFAULT`) quota. Since some programs may have larger memory requirements than others, a program can call the `$ADJWSL` system service to dynamically increase the process's working set limit. When the additional pages are no longer needed in the working set, the program can call the `$ADJWSL` system service to decrease the working set limit. It can also call the `$PURGWS` system service to remove pages no longer in use from the working set.

Under some circumstances, an image may not want certain pages to be paged out at all; in this case, the image can lock these pages in the working set with the Lock Pages in Working Set (`$LKWSET`) system service. As long as the process's working set is in memory, these pages cannot be paged out

until they are explicitly unlocked with the Unlock Pages in Working Set (\$ULWSET) system service.

11.5 Process Swapping

The operating system balances the needs of all the processes that are currently executing, providing each with the system resources it requires on an as-needed basis. The memory management routines balance the process's memory requirements. Thus, the sum of the working sets for all processes that are currently in physical memory is called the balance set.

When a process whose working set is in memory becomes inactive—for example, to wait for an I/O request or to hibernate—the entire or part of the working set may be removed from memory to provide space for another process's working set to be brought in for execution.

The working set may be removed in two ways:

- Partially—also called swapper trimming. Pages are removed from the working set of the target process so the amount of pages in the working set are fewer, but none of the pages are swapped out of the working set.
- Entirely—called swapping. All pages are swapped out of the working set.

When a process is swapped out of the balance set, all the pages (both modified and unmodified) of its working set are swapped, including any pages that had been locked in the working set.

It is possible for a privileged process to lock its entire working set in the balance set. While pages can still be paged in and out of the working set, the process remains in memory even when it is inactive. To lock itself in the balance set, the process issues the Set Process Swap Mode (\$SETSWM) system service, as follows:

```
$SETSWM_8 SWPFLG=#1
```

This call to \$SETSWM disables process swap mode. Swap mode can also be disabled by setting the appropriate bit in the STSFLG argument to the Create Process (\$CREPRC) system service; however, you must have the PSWAPM privilege to alter process swap mode.

A process can also lock pages in memory with the Lock Pages in Memory (\$LCKPAG) system service. When a page is locked in memory with this service, the page remains in memory even when the remainder of the process's working set is swapped out of the balance set. This system service can be useful in special circumstances, for example, for routines that perform I/O operations to slow devices or graphics devices.

Pages locked in memory can be unlocked with the Unlock Pages in Memory (\$ULKPAG) system service. The user privilege PSWAPM is required to issue the \$LCKPAG or \$ULKPAG service.

Memory Management Services

Sections

11.6 Sections

A section is a disk file or a portion of a disk file containing data or instructions that can be brought into memory and made available to a process for manipulation and execution. A section can also be one or more consecutive page frames in physical memory or I/O space; such sections, which require you to specify page frame number mapping, are discussed in Section 11.6.14.

Sections are either private or global (shared).

- Private sections are accessible only by the process that creates them. A process can define a disk data file as a section, map it into its virtual address space, and manipulate it.
- Global sections can be shared by more than one process. One copy of the global section resides in physical memory, and each process sharing it refers to the same copy. A global section can contain shareable code or data that can be read, or read and written, by more than one process. Global sections are either temporary or permanent and can be defined for use within a group or on a system-wide basis. Global sections can be either mapped to a disk file or created as a global page-file section.

When modified pages in writable disk file sections are paged out of memory during image execution, they are written back into the section file, rather than into the paging file, as is the normal case with files. (However, copy-on-reference sections are not written back in to the section file.)

The use of disk file sections involves two distinct operations:

- 1 The creation of a section defines a disk file as a section and informs the system what portions of the file contain the section.
- 2 The mapping of a section makes the section available to a process and establishes the correspondence between virtual blocks in the file and specific addresses in the process's virtual address space.

The Create and Map Section (\$CRMPSC) system service creates and/or maps a private section or a global section. Since a private section is used only by a single process, creation and mapping are simultaneous operations. In the case of a global section, one process can create a permanent global section and not map to it; other processes can map to it. A process can also create and map a global section in one operation.

The following sections describe creating, mapping, and using disk file sections. In each case, operations and requirements that are common to both private sections and global sections are described first, followed by additional notes and requirements for the use of global sections. Section 11.6.14 discusses special requirements for page frame sections; Section 11.6.8 discusses global page-file sections.

11.6.1 Creating Sections

The following steps are involved in creating disk file sections.

- 1 Opening or creating the disk file containing the section
- 2 Defining which virtual blocks in the file comprise the section
- 3 Defining the characteristics of the section

11.6.2 Opening the Disk File

Before a file can be used as a section, it must be opened using VAX RMS. The following example shows the VAX RMS file access block (\$FAB) and \$OPEN macros used to open the file, and the channel specification to the \$CRMPSC system service necessary for reading an existing file.

```
SECFAB: $FAB      FNM=<SECTION.TST>, ; file access block
                  FOP=UFO
                  RTV= -1

                  .
                  .
                  .
$OPEN  FAB=SECFAB
$CRMPSC_S -
        CHAN=SECFAB+FAB$L_STV,...
```

The file options parameter (FOP) indicates that the file is to be opened for user I/O; this option is required so that VAX RMS assigns the channel using the access mode of the caller. VAX RMS returns the channel number on which the file is accessed; this channel number is specified as input to the \$CRMPSC system service (**chan** argument). The same channel number can be used for multiple create and map section operations.

The equation, RTV= -1 tells the file system to keep all of the pointers to be mapped in memory at all times. Storage for these pointers is charged to BYTLM quota which means that opening a badly fragmented file can fail with an EXBYTLM failure status. Too many fragmented sections may cause the byte limit to be exceeded.

The file may be a new file that is to be created while it is in use as a section. In this case, use the \$CREATE macro to open the file. If you are creating a new file, the file access block (FAB) for the file must specify an allocation quantity (ALQ parameter).

\$CREATE can also be used to open an existing file; if the file does not exist, it will be created. The following example shows the required fields in the FAB for the conditional creation of a file.

```
GBLFAB: $FAB      FNM=<GLOBAL.TST>, -
                  ALQ=4, -
                  FAC=PUT, -
                  FOP=<UFO,CIF,CBT>, -
                  SHR=<PUT,UPI>

                  .
                  .
                  .
$CREATE FAB=GBLFAB
```

When the \$CREATE macro is invoked, it creates the file GLOBAL.TST if the file does not currently exist. The CBT (contiguous-best-try) option requests that if possible, the file be contiguous. Although it is not required that section files be contiguous, better performance can result if they are.

Memory Management Services

Sections

11.6.3 Defining the Section Extents

Once the file is successfully opened, the \$CRMPSC system service can create a section from the entire file, or from only certain portions of it. The following arguments to \$CRMPSC define the extents of the file that comprises the section:

- **pagcnt** (page count). This argument is required; it indicates the number of virtual blocks that will be mapped. These blocks correspond to pages in the section.
- **vbn** (virtual block number). This argument defines the number of the virtual block in the file that is the beginning of the section. It is an optional argument. If you do not specify this argument, the value 1 is passed (the first virtual block in the file is the beginning of the section). If you have specified physical page frame number mapping, the **vbn** argument specifies the starting page frame number.

11.6.4 Defining the Section Characteristics

The **flags** argument to the \$CRMPSC system service defines the following section characteristics:

- Whether it is a private section or a global section (the default is to create a private section)
- How the pages of the section are to be treated when they are copied into physical memory or when a process refers to them. The pages in a section can be:
 - Read/write or read-only
 - Created as demand-zero pages or as copy-on-reference pages, depending on how the processes are going to use the section and whether the file contains any data (see Section 11.6.9, Section Paging).
- Whether the section is to be mapped to a disk file or to specific physical page frames (Section 11.6.14 discusses physical page frame sections).

Table 11-2 shows the flag bits that must be set for specific characteristics.

Memory Management Services

Sections

Table 11-2 Flag Bits to Set for Specific Section Characteristics

Correct Flag Combinations	Section to be Created				
	Private	Global	PFN Private	PFN Global	Shared Memory
SEC\$M_GBL	0	1	0	1	1
SEC\$M_CRF	Optional	Optional	0	0	0
SEC\$M_DZRO	Optional	Optional	0	0	Optional
SEC\$M_WRT	Optional	Optional	Optional	Optional	Optional
SEC\$M_PERM	Not used	Optional	Optional	1	1
SEC\$M_SYSGBL	Not used	Optional	Not used	Optional	Optional
SEC\$M_PFNMAP	0	0	1	1	0
SEC\$M_EXPREG	Optional	Optional	Optional	Optional	Optional
SEC\$M_PAGFIL	0	1	0	0	0

When specifying section characteristics, the following restrictions hold:

- Global sections cannot be both demand-zero and copy-on-reference.
- Demand-zero sections must be writeable.
- Shared memory private sections are not allowed.

11.6.5 Defining Global Section Characteristics

If the section is a global section, it must be assigned a character string name (**gsdnam** argument) so that other processes can identify it when they map it. The format of this character string name is explained in Section 11.6.5.1.

The **flags** argument specifies the type of global section.

- Group temporary (the default)
- Group permanent
- System temporary
- System permanent

Group global sections can be shared only by processes executing with the same group number. The name of a group global section is implicitly qualified by the group number of the process that created it. When other processes map it, their group numbers must match.

A temporary global section is automatically deleted when no processes are mapped to it, but a permanent global section remains in existence even when no processes are mapped to it. A permanent global section must be explicitly marked for deletion with the Delete Global Section (\$DGBLSC) system service.

The user privileges PRMGBL and SYSGBL are required to create permanent group global sections or system global sections (temporary or permanent), respectively.

Memory Management Services

Sections

A system global section is available to all processes in the system.

Optionally, a process creating a global section can specify a protection mask (**prot** argument) restricting all access or a type of access (read, write, execute, delete) to other processes.

11.6.5.1

Global Section Name

The **gsdnam** argument specifies a descriptor that points to a character string with the following format:

[shared-memory-name:]global-section-name

Arguments

shared-memory-name

Identifies the global section to be created, mapped, or deleted as within the named memory that is shared by multiple processors. The name of this memory was specified at system generation time. For example, the string **SHRMEM\$1:GSDATA** identifies a global section named **GSDATA** located in the shared memory named **SHRMEM\$1**.

If this part of the string is not included and the section is being mapped or deleted, the system tries to find the specified global section first in local memory and then in shared memory units (in the order in which they were connected).

global-section-name

The name assigned to the global section. You may choose any valid name, from 1 to 43 characters; however, all processes mapping to the same global section must specify the same name.

If you wish, you can include both the **shared-memory-name** and the **global-section-name** for a global section in memory shared by multiple processors. However, if you want to use existing programs without recompiling or relinking, or if you want the program to run regardless of whether the section is in local memory or shared memory, you can specify only a **global-section-name** and have the system translate it to a complete specification. The system attempts to perform logical name translation of the string specified by the **gsdnam** argument in the following manner.

- The current name string is searched for a colon. If a colon is found within the current name string, the global section is assumed to be located in shared memory and translation proceeds in the following manner.
 - 1 The portion of the current name string to the right of the colon is placed in the **global-section-name** buffer. The portion of the current name string to the left of the colon becomes the new current name string.
 - 2 **GBL\$** is prefixed to the current name string and the result is subjected to logical name translation.
 - 3 If the result contains a logical name, steps 1 and 2 are repeated until translation does not succeed or until the number of translations performed exceeds the number specified by the **SYSGEN** parameter **LNM\$C_MAXDEPTH**.
 - 4 The **GBL\$** prefix is stripped from the current name string that could not be translated. This name becomes the shared memory name. The **global-section-name** is the current string contained in the **global-section-name** buffer.

Memory Management Services

Sections

- If the global section is located in local memory, translation proceeds in the following manner.
 - 1 GBL\$ is prefixed to the current name string and the result is subjected to logical name translation.
 - 2 If the result is a logical name, step 1 is repeated until translation does not succeed or until the number of translations performed exceeds the number specified by the SYSGEN parameter LNM\$C_MAXDEPTH.
 - 3 The GBL\$ prefix is stripped from the current name string that could not be translated. This current string is the global-section-name.

For example, assume that you have made the following logical name assignment:

```
$ DEFINE GBL$GSDATA SHRMEM$1:GSDATA
```

Your program contains the following statements:

```
NAMEDESC:
    .ASCID  /GSDATA/      ; descriptor for logical name
                        ; of section
    .
    $CRMPSC_8 -
        GSDNAM=NAMEDESC,...
```

The following logical name translation takes place.

- 1 GBL\$ is prefixed to SHRMEM\$1.
- 2 GBL\$GSDATA is translated to SHRMEM\$1:GSDATA. (There is translation for GBL\$SHRMEM\$1, so no further translation is successful. When logical name translation fails, the string is passed to the service.)

There are three exceptions to the logical name translation method discussed in this section:

- If the name string starts with an underscore (_), VAX/VMS strips the underscore and considers the resultant string to be the actual name (that is, no further translation is performed).
- If the name string is the result of a logical name translation, then the name string is checked to see if it has the "terminal" attribute. If the name string is marked with the "terminal" attribute, VAX/VMS considers the resultant string to be the actual name (that is, no further translation is performed).
- If the global section has a name in the format name_nnn, VAX/VMS first strips the underscore and the digits (nnn), then translates the resultant name according to the sequence discussed in this section, and finally reappends the underscore and digits. The system uses this method in conjunction with known images and shared files installed by the system manager.

Memory Management Services

Sections

11.6.6 Mapping Sections

When you call the \$CRMPSC system service to create and/or map a section, you must provide the service with a range of virtual addresses (**inadr** argument) into which the section is to be mapped.

If you know specifically which pages the section should be mapped into, you provide these addresses in a two-longword array. For example, to map a private section of 10 pages into virtual pages 10 through 19 of the program region, specify the input address array as follows:

```
MAPRANGE:
    .LONG    ~X1400          ; address (hex) of page 10
    .LONG    ~X2300          ; address (hex) of page 19
```

However, you do not need to know the explicit addresses to provide an input address range. If you simply want the section mapped into the first available virtual address range in the program (P0) or control (P1) region, you can specify the SEC\$M_EXPREG flag bit in the **flags** argument. In this case, the addresses specified by the **inadr** argument simply control whether the service finds the first available space in the program or control region. The value specified or defaulted for the **pagcnt** argument determines the number of pages mapped. The following example shows part of a program used to map a section at the current end of the program region.

```
MAPRANGE:
    .LONG    ~X200           ; any program (P0) region address
    .LONG    ~X200           ; any P0 address (can be same)
RETRANGE:
    .BLKL    2               ; address range returned here

$CRMPSC_S -
    INADR=MAPRANGE, -
    RETADR=RETRANGE, -
    FLAGS=<SEC$M_EXPREG>,...
```

The addresses specified do not have to be currently in the process's virtual address space. The \$CRMPSC system service creates the required virtual address space during the mapping of the section. If you specify the **retadr** argument, the service returns the range of addresses actually mapped.

Once a section has been successfully mapped, the image can refer to the pages using one of the following:

- A base register or pointer and predefined symbolic offset names
- Labels defining offsets of an absolute program section or structure

The following example shows part of a program used to create and map a process section.

Memory Management Services

Sections

```

SECFAB: $FAB      FNM=<SECTION.TST>, -
                  FOP=UFO, -
                  FAC=PUT, -
                  SHR=<GET,PUT>

;
MAPRANGE:
    .LONG    ^X1400          ; first page
    .LONG    ^X2300          ; last page
RETRANGE:
    .BLKL    1              ; first page mapped
ENDRANGE:
    .BLKL    1              ; last page mapped

;
$OPEN    FAB=SECFAB          ; open section file
BSBW     ERROR
$CRMPSC _S -
    INADR=MAPRANGE, -        ; input address array
    RETADR=RETRANGE, -      ; output array
    PAGCNT=#4, -            ; map four pages
    FLAGS=#SEC$M_WRT, -     ; read/write section
    CHAN=SECFAB+FAB$L_STV   ; channel number
BSBW     ERROR
MOVL     RETRANGE,R6         ; point to start of section

```

Notes on Example:

- 1 The OPEN macro opens the section file defined in the file access block SECFAB. (The FOP parameter to the \$FAB macro must specify the UFO option.)
- 2 The \$CRMPSC system services uses the addresses specified at MAPRANGE to specify an input range of addresses into which the section will be mapped. The **pagcnt** argument requests that only four pages of the file be mapped.
- 3 The **flags** argument requests that the pages in the section have read/write access. The symbolic flag definitions for this argument are defined in the \$SECDEF macro. Note that the file access field (FAC parameter) in the FAB also indicates that the file is to be opened for writing.
- 4 When \$CRMPSC completes, the addresses of the four pages that were mapped are returned in the output address array at RETRANGE. The address of the beginning of the section is placed in general register 6, which serves as a pointer to the section.

11.6.7 Mapping Global Sections

A process that creates a global section can map to it when it creates it. Then, other processes can map it by calling the Map Global Section (\$MGBLSC) system service.

When a process maps a global section, it must specify the global section name assigned to the section when it was created, whether it is a group or system global section, and whether it desires read-only or read/write access. The process may also specify:

- A version identification (**indent** argument), indicating the version number of the global section (when multiple versions exist) and whether more recent versions are acceptable to the process.
- A relative page number (**relpag** argument), specifying the page number, relative to the beginning of the section, to begin mapping the section. In this way, processes can use only portions of a section. Additionally, a process can map a piece of a section into a particular address range and

Memory Management Services

Sections

subsequently map a different piece of the section into the same virtual address range.

To specify that the global section being mapped is located in physical memory that is being shared by multiple processors, you can include the shared memory name in the **gsdnam** argument character string (see Section 11.6.5.1). A demand-zero global section in memory shared by multiple processors must be mapped when it is created.

Cooperating processes can both issue a \$CRMPSC system service to create and map the same global section. The first process to call the service actually creates the global section; subsequent attempts to create and map the section result only in mapping the section for the caller. The successful return status code SS\$_CREATED indicates that the section did not already exist when the \$CRMPSC system service was called. If the section did exist, the status code SS\$_NORMAL is returned.

The example in Section 11.6.9 shows one process (ORION) creating a global section and a second process (CYGNUS) mapping the section.

11.6.8 Global Page-File Sections

Global page-file sections are used to store temporary data in a global section. A global page-file section is a section of virtual memory that is not mapped to a file. The section can be deleted when processes have finished with it. Contrast this with demand-zero pages where no initialization is necessary, but the pages are saved in a file. The SYSGEN parameter GBLPAGFIL controls the total number of global page-file pages in the system.

To create a global page-file section, you must set the flag bits SEC\$_M_GBL and SEC\$_M_PAGFIL in the **flags** argument to the Create and Map Section (\$CRMPSC) system service. The channel (**chan** argument) must be zero.

You cannot specify the flag bit SEC\$_M_CRF with the flag bit SEC\$_M_PAGFIL.

11.6.9 Section Paging

The first time that an image executing in a process refers to a page that was created during the mapping of a disk file section, the page is copied into physical memory. The address of the page in the process's virtual address space is mapped to the physical page. During the execution of the image, normal paging can occur; however, pages in sections are not written into the page file when they are paged out, as is the normal case. Rather, if they have been modified, they are written back into the section file on disk. The next time a page fault occurs for the page, the page is brought back from the section file.

However, if the pages in a section were defined as demand-zero pages or copy-on-reference pages when the section was created, the pages are treated differently.

- If the call to \$CRMPSC requested that pages in the section be treated as demand-zero pages, these pages are initialized to zeros when they are created in physical memory. If the file is either a new file that is being created as a section or a file that is being completely rewritten, demand-zero pages provide a convenient way of initializing the pages. The pages are paged back into the section file.

Memory Management Services

Sections

- When the virtual address space is deleted, all unreferenced pages are written back to the file as zeros. This causes the file to be initialized, no matter how few pages were modified.
- If the call to \$CRMPSC requested that pages in the section be copy-on-reference pages, each process that maps to the section receives its own copy of the section, on a page-by-page basis from the file, as it refers to them. These pages are never written back into the section file, but are paged to the paging file as needed.

In the case of global sections, more than one process can be mapped to the same physical pages. If these pages need to be paged out or written back to the disk file defined as the section, these operations are done only when the pages are not in the working set of any process.

```
Process ORION
FLGCLUSTER:      ; descriptor for common event flag cluster name
  .ASCID /FLAG_CLUSTER/
GLOBALSEC:        ; descriptor for global section name
  .ASCID /GLOBAL_SECTION/
;
FLGSET = 06      ; flag number to associate and set
GBLFAB: $FAB     FNM=<GLOBAL.TST>, -
                FOP=<UFO,CIF,CST>,-
                ALQ=4, -
                FAC=PUT
```

```
① $ASCEFC_S -
    EFN=$FLGSET, -
    NAME=FLGCLUSTER
BSBW ERROR
② $CRMPSC_S -      ; create global section
    GSDNAM=GLOBALSEC,-
    FLAGS=$SEC$M_WRT!SEC$M_GBL,...
BSBW ERROR
$SETEF_S -        ; set common event flag
    EFN=$FLGSET
```

```
Process CYGNUS
CLUSTER:
  .ASCID /FLAG_CLUSTER/ ; cluster name descriptor
SECTION:
  .ASCID /GLOBAL_SECTION/ ; section name descriptor
FLGSET = 06
```

```
③ $ASCEFC_S -
    EFN=$FLGSET, -
    NAME=CLUSTER
BSBW ERROR
$WAITFR_S -
    EFN=$FLGSET
BSBW ERROR
$MGBLSC_S -
    INADR=MAPRANGE, -
    RETADR=RETRANGE,-
    FLAGS=$SEC$M_GBL,- ; global section
    GSDNAM=SECTION ; section name
BSBW ERROR
```

Notes on Example:

- ① The processes ORION and CYGNUS are in the same group. Each process first associates with a common event flag cluster named FLAG_CLUSTER to use common event flags to synchronize its use of the section.

Memory Management Services

Sections

- ORION creates the global section named GLOBAL_SECTION, specifying section flags that indicate that it is a global section (SEC\$M_GBL) and that it has read/write access. Input and output address arrays, the page count parameter and the channel number arguments are not shown; procedures for specifying them are the same as shown in the example shown above.
- The process CYGNUS associates with the common event flag cluster and waits for the flag defined as FLGSET. ORION sets this flag when it has completed creating the section. To map the section, CYGNUS specifies the input and output address arrays, the flag indicating that it is a global section, and the global section name. In this example, the number of pages mapped is the same as that specified by the creator of the section.

11.6.10 Reading and Writing Data Sections

Read/write sections provide a way for a process or cooperating processes to share data files in virtual memory.

The sharing of global sections may involve application-dependent synchronization techniques. For example, one process can create and map to a global section in read/write fashion; other processes can map to it in read-only fashion and interpret data written by the first process. Or, two or more processes can write to the section concurrently. (In this case, the application must provide the necessary synchronization and protection.)

When a file has been updated, the process or processes can release (or unmap) the section. The modified pages are then written back into the disk file that is defined as a section.

When this is done, the revision number of the file is incremented, and the version number of the file remains unchanged. A full directory listing indicates the revision number of the file and the date and time that the file was last updated.

11.6.11 Releasing and Deleting Sections

A process unmaps a section by deleting the virtual addresses in its own virtual address space to which it has mapped the section. If a return address range was specified to receive the virtual addresses of the mapped pages, this address range can be used as input to the Delete Virtual Address Space (\$DELTVA) system service as follows:

```
$DELTVA_S INADR=RETRANGE
```

When a process unmaps a private section, the section is deleted; that is, all control information maintained by the system is deleted. A temporary global section is deleted when all processes that have mapped to it have unmapped it. Permanent global sections are not deleted until they are specifically marked for deletion with the Delete Global Section (\$DGBLSC) system service; they are then deleted when no more processes are mapped.

Note that deleting the pages occupied by a section does not delete the section file, but rather cancels the process's association with the file. Moreover, when a process deletes pages mapped to a read/write section and no other processes are mapped to it, all modified pages are written back into the section file.

When a section has been deleted, the channel assigned to it can be deassigned. The process that created the section can deassign the channel (with the Deassign I/O Channel system service) as follows:

```
$DASSGN_S CHAN=GBLFAB+FAB$L_STV
```

11.6.12 Writing Back Sections

Since read/write sections are normally not updated on disk until the physical pages they occupy are paged out, or until all processes referring to the section have unmapped it, a process may want to ensure that all modified pages are successfully written back into the section file at regular intervals.

The Update Section File on Disk (\$UPDSEC) system service writes the modified pages in a section into the disk file. The \$UPDSEC system service is described in Part II.

11.6.13 Image Sections

Global sections can contain shareable code. The operating system uses global sections to implement shareable code as follows:

- 1 The object module containing code to be shared is linked to produce a shareable image. The shareable image is not, in itself, executable. It contains a series of sections, called image sections.
- 2 A user links private object modules with the shareable image to produce an executable image. No code or data from shareable image is put into executable image.
- 3 The system manager uses the INSTALL command to create a permanent global section from the shareable image file, making the image sections available for sharing.
- 4 When the user runs the executable image, VAX/VMS automatically maps the global sections created by the INSTALL command into the virtual address space of the user's process.

For details on how to create and identify shareable images and how to link them with private object modules, see the *VAX/VMS Linker Reference Manual*. For information on installing shareable images and making them available for sharing as global sections, see the *Guide to VAX/VMS System Management and Daily Operations*.

11.6.14 Page Frame Sections

A page frame section is one or more contiguous pages of physical memory or I/O space that have been mapped as a section. One use of page frame sections is to map to an I/O page, thus allowing a process to read device registers. A process mapped to an I/O page can also connect to a device interrupt vector.

A page frame section differs from a disk file section in that it is not associated with a particular disk file and is not paged. However, it is similar to a disk file section in most other respects: you create, map, and define the extent and characteristics of a page frame section in essentially the same manner as you do a disk file section.

Memory Management Services

Sections

To create a page frame section, you must specify page frame number mapping by setting the SEC\$M_PFNMAP flag bit in the **flags** argument to the Create and Map Section (\$CRMPSC) service. The **vbn** argument is now used to specify the first page frame to be mapped instead of the first virtual block. You must have the user privilege PFNMAP to create or delete a page frame section, but not to map to an existing one.

Because this type of section is not associated with a disk file, the **relpag**, **chan**, and **pfc** arguments to the \$CRMPSC service are not used in creating or mapping a page frame section. For the same reason, the SEC\$M_CRF (copy-on-reference) and SEC\$M_DZRO (demand-zero) bit settings in the **flags** argument do not apply. Pages in page frame sections are not written back to any disk file (including the paging file).

Use caution in working with page frame sections. If you permit write access to the section, each process that writes to the section does so at its own risk. Serious errors can occur if a process writes incorrect data or writes to the wrong page, especially if the page is also mapped by the system or by another process. Thus, the security of a system can be violated or damaged by any user having the PFNMAP privilege.

11.7 Example Using Memory Management System Services

```
!This is the program that creates the global section.
! Define global section flags
INCLUDE '($SECDDEF)'
! Mask for section flags
INTEGER SEC_MASK
! Logical unit number for section file
INTEGER INFO_LUN
! Channel number for section file
! (returned from useropen routine)
INTEGER SEC_CHAN
COMMON /CHANNEL/ SEC_CHAN
! Length for the section file
INTEGER SEC_LEN
! Data for the section file
CHARACTER*12 DEVICE,
2      PROCESS
CHARACTER*6 TERMINAL
COMMON /DATA/ DEVICE,
2      PROCESS,
2      TERMINAL
! Location of data
INTEGER PASS_ADDR (2),
2      RET_ADDR (2)
! Two common event flags
INTEGER REQUEST_FLAG,
2      INFO_FLAG
DATA REQUEST_FLAG /70/
DATA INFO_FLAG /71/
! User-open routines
INTEGER UFO_CREATE
EXTERNAL UFO_CREATE

! Open the section file
STATUS = LIB$GET_LUN (INFO_LUN)
IF (.NOT. STATUS) CALL LIB$SIGNAL(XVAL(STATUS))
SEC_MASK = SEC$M_WRT .OR. SEC$M_DZRO .OR. SEC$M_GBL
! (last element - first element + size of last element + 511)/512
SEC_LEN = ( (%LOC(TERMINAL) - %LOC(DEVICE) + 6 + 511)/512 )
OPEN (UNIT=INFO_LUN,
```

Memory Management Services

Example Using Memory Management System Services

```

2  FILE='INFO.TMP',
2  STATUS='NEW',
2  INITIALSIZE = SEC_LEN,
2  USEROPEN = UFO_CREATE)
! Free logical unit number and map section
CLOSE (INFO_LUN)

! Get location of data
PASS_ADDR (1) = %LOC (DEVICE)
PASS_ADDR (2) = %LOC (TERMINAL)
STATUS = SYS$CRMPSC (PASS_ADDR, !Address of section
2  RET_ADDR, !Addresses mapped
2
2  %VAL(SEC_MASK), !Section mask
2  'GLOBAL_SEC', !Section name
2
2  %VAL(SEC_CHAN), !I/O channel
2  ...)
IF (.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))

! Create the subprocess
STATUS = SYS$CREPRC (,
2  'GETDEVINF', ! Image
2
2  'GET_DEVICE', ! Process name
2  %VAL(4)... ! Priority
2
IF (.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))

! Write data to section
DEVICE = '$FLOPPY1'

! Get common event flag cluster and set flag
STATUS = SYS$ASCEFC (%VAL(REQUEST_FLAG),
2  'CLUSTER',...)
IF (.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))
STATUS = SYS$SETEF (%VAL(REQUEST_FLAG))
IF (.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))

! When GETDEVINF has the information, INFO_FLAG is set
STATUS = SYS$WAITFR (%VAL(INFO_FLAG))
IF (.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))


!This is the program that maps to the global section
!created by the previous program.

! Define section flags
INCLUDE '($SECDDEF)'
! Mask for section flags
INTEGER SEC_MASK
! Data for the section file
CHARACTER*12 DEVICE,
2  PROCESS
CHARACTER*6 TERMINAL
COMMON /DATA/ DEVICE,
2  PROCESS,
2  TERMINAL

! Location of data
INTEGER PASS_ADDR (2),
2  RET_ADDR (2)

! Two common event flags
INTEGER REQUEST_FLAG,
2  INFO_FLAG
DATA REQUEST_FLAG /70/
DATA INFO_FLAG /71/


! Get common event flag cluster and wait
! for GBL1.FOR to set REQUEST_FLAG
STATUS = SYS$ASCEFC (%VAL(REQUEST_FLAG),
2  'CLUSTER',...)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))

```

Memory Management Services

Example Using Memory Management System Services

```
STATUS = SYS$WAITFR (%VAL(REQUEST_FLAG))
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
! Get location of data
PASS_ADDR (1) = %LOC (DEVICE)
PASS_ADDR (2) = %LOC (TERMINAL)
! Set write flag
SEC_MASK = SEC$M_WRT
! Map the section
STATUS = SYS$MGBLSC (PASS_ADDR, ! Address of section
2 RET_ADDR, ! Address mapped
2
2 %VAL(SEC_MASK), ! Section mask
2 'GLOBAL_SEC',,) ! Section name
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
! Call GETDVI to get the process id of the
! process that allocated the device, then
! call GETJPI to get the process name and terminal
! name associated with that process id.
! Set PROCESS equal to the process name and
! set TERMINAL equal to the terminal name.

! After information is in GLOBAL_SEC
STATUS = SYS$SETEF (%VAL(INFO_FLAG))
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
END
```

In this example, there are two programs communicating through a global section. The first program creates and maps a global section (by using the \$CRMPSC system service), then writes a device name to the section. This program also defines the device terminal and process names and sets the event flags which synchronize the processes.

The second program maps the section (by using the \$MGBLSC system service), reads the device name and the process that allocated the device and any terminal allocated to that process. This program also writes the process named to the terminal global section where they can be read by the first program.

The common event cluster is used to synchronize access to the global section. The first program sets REQ_FLAG to indicate the device name is in the section. The second program sets INFO_FLAG to indicate that the process and terminal names are available.

Data in a section must be page aligned. This is the option file used at LINK time that causes the data in the common area named DATA to be page aligned.

PSECT_ATTR = DATA, PAGE

Before executing the first program, you would have to write a user open routine that would set the user open bit (FAB\$V_UFO) of the FAB options longword (FAB\$L_FOP). In addition, the user-open routine would read the channel number that the file is opened on from the status longword (FAB\$L_STV) and return that channel number to the main program by using a common block (CHANNEL in this example).

12 Lock Management Services

The VAX/VMS lock management system services allow cooperating processes to synchronize their access to shared resources. This is accomplished by providing a common data area in which processes can lock a specified resource by name. All processes that access the resources must use the VAX/VMS lock management services, or they are not effective.

To synchronize access to resources, the lock management services provide a queuing mechanism that allows processes to wait in a queue until a particular resource is available.

The Enqueue Lock Request (\$ENQ) system service is used to make lock requests and the Dequeue Lock Request (\$DEQ) system service is used to cancel lock requests. The Get Lock Information (\$GETLKI) system service is used to get information about existing locks.

12.1 Concepts of Resources and Locks

A resource can be any entity on VAX/VMS (for example, files, data structures, databases, and executable routines). When two or more processes access the same resource, it is often necessary to control their access to the resource. It is not desirable to have one process reading the resource while another process writes new data; a writer can quickly invalidate anything being read by a reader. The lock management system services allow processes to associate a name with a resource and request access to that resource. Lock modes enable processes to indicate how they want to share access with other processes.

To use the lock management system services, a process must request access to a resource (request a lock) using the Enqueue Lock Request (\$ENQ) system service. There are three required arguments to the \$ENQ system service for new locks.

- A resource name. The lock management services use the resource name to look for other lock requests that use the same name.
- The lock mode to be associated with the requested lock. The lock mode indicates how the process wants to share the resource with other processes.
- The address of a lock status block. The lock status block receives the completion status for a lock request and the lock identification. The lock identification is used to refer to a lock request once it has been queued.

The lock management services compare the lock mode of the newly requested lock to the lock modes of other locks with the same resource name.

- If no other process has a lock on the resource, the new lock is granted.
- If another process has a lock on the resource and the mode of the new request is compatible with the existing lock, the new lock is granted.

Lock Management Services

Concepts of Resources and Locks

- If another process already has a lock on the resource and the mode of the new request is not compatible with the lock mode of the existing lock, the new request is placed in a queue where it waits until the resource becomes available. When the resource becomes available, the process is notified that it can access the resource (the lock is granted).

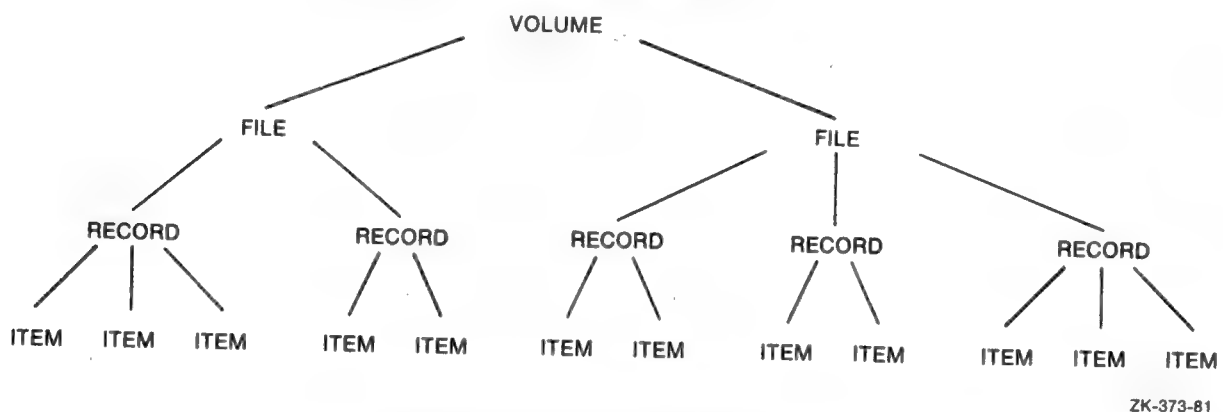
Processes can also use the \$ENQ system service to change the lock mode of a lock. This is called a lock conversion.

12.1.1 Granularity

Many resources can be divided into smaller parts. So long as a part of a resource can be identified by a resource name, the part can be locked. The term granularity describes the part of the resource being locked.

Figure 12-1 below depicts a model of a database. The database is divided into areas, which in turn are subdivided into records. The records are further divided into items.

Figure 12-1 Model Database



ZK-373-81

The processes that request locks on this database must decide whether to lock the whole database, an area in the database, a record, or a single item. Locking the entire database is considered locking at a coarse granularity; locking a single item is considered locking at a fine granularity.

12.1.2 Resource Names

The lock management system services refer to each resource by a name composed of four parts. For two resources to be considered the same, these four parts must be identical for each resource.

- A name specified by the caller
- The caller's access mode
- The caller's UIC group number (unless the resource is system-wide)
- The identification of the lock's parent (optional)

Lock Management Services

Concepts of Resources and Locks

The name specified by the process represents the resource being locked. Other processes that need to access the resource must refer to it using the same name. The correlation between the name and the resource is simply a convention agreed upon by the cooperating processes.

The access mode is determined by the caller's access mode, unless a less privileged mode is specified in the call to the \$ENQ system service. Access modes, their numeric values, and symbolic names are discussed in Section 2.2.1.3.

Resources can be group-specific or system-wide. The default is for resource names to be qualified by the group number of the calling process's UIC. System-wide locks are defined by setting a flag bit in the call to the \$ENQ system service. The user privilege SYSLOCK is required to request system-wide locks from user or supervisor mode. No additional privilege is required to request system-wide locks from executive or kernel mode.

When a lock request is queued, it can specify the identification of a parent lock, at which point it becomes a sublock. However, the parent lock must be granted or the lock request is not accepted. This enables a process to lock a resource at different degrees of granularity.

12.1.3 Choosing a Lock Mode

The mode of a lock determines whether or not the resource can be shared with other lock requests. The six lock modes are listed below.

Mode Name	Meaning
LCK\$K_NLMODE	Null mode. This mode grants no access to the resource; the null mode is typically used as an indicator of interest in the resource, or as a place holder for future lock conversions.
LCK\$K_CRMODE	Concurrent read. This mode grants read access to the resource and allows sharing of the resource with other readers. The concurrent read mode is generally used when additional locking is being performed at a finer granularity with sublocks, or to read data from a resource in an "unprotected" fashion (allowing simultaneous writes to the resource).
LCK\$K_CWMODE	Concurrent write. This mode grants write access to the resource and allows sharing of the resource with other writers. The concurrent write mode is typically used to perform additional locking at a finer granularity, or to write in an "unprotected" fashion.
LCK\$K_PRMODE	Protected read. This mode grants read access to the resource and allows the resource to be shared with other readers. No writers are allowed access to the resource. This is the traditional "share lock."

Lock Management Services

Concepts of Resources and Locks

Mode Name	Meaning
LCK\$K_PWMODE	Protected write. This mode grants write access to the resource and allows the resource to be shared with concurrent read mode readers. No other writers are allowed access to the resource. This is the traditional "update lock."
LCK\$K_EXMODE	Exclusive. The exclusive mode grants write access to the resource and prevents the resource from being shared with any other readers or writers. This is the traditional "exclusive lock."

12.1.4 Levels of Locking and Compatibility

Locks that allow the process to share a resource are called low-level locks; locks that allow the process almost exclusive access to a resource are called high-level locks. Null and concurrent read mode locks are considered low-level locks; protected write and exclusive mode locks are considered high-level. The lock modes from lowest to highest level access modes are: null, concurrent read, concurrent write, protected read, protected write, and exclusive. The concurrent write and protected read modes are considered to be of equal level.

Locks that can be shared with other locks are said to have compatible lock modes. Higher-level lock modes are less compatible with other lock modes than are lower-level lock modes. Table 12-1 shows the compatibility of the lock modes.

12.1.5 Lock Management Queues

A lock on a resource can be in one of three states.

GRANTED The lock request has been granted.

WAITING The lock request is waiting to be granted.

CONVERSION The lock request has been granted at one mode and is waiting to be granted a higher lock mode.

A queue is associated with each of the three states (see Figure 12-2).

Table 12-1 Compatibility of Lock Modes

Mode of Requested Lock	Mode of Currently Granted Locks					
	NL	CR	CW	PR	PW	EX
NL	Yes	Yes	Yes	Yes	Yes	Yes
CR	Yes	Yes	Yes	Yes	Yes	No
CW	Yes	Yes	Yes	No	No	No
PR	Yes	Yes	No	Yes	No	No
PW	Yes	Yes	No	No	No	No
EX	Yes	No	No	No	No	No

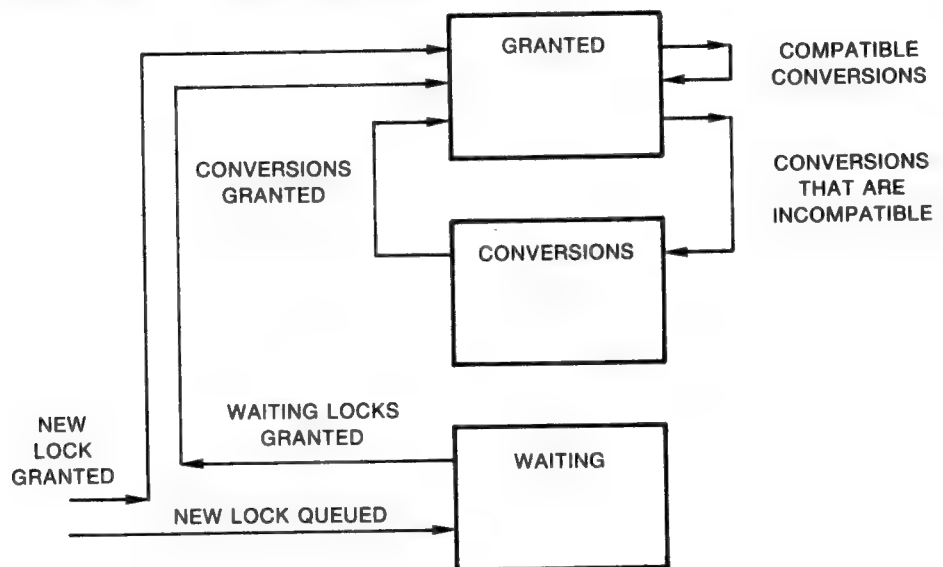
Lock Management Services

Concepts of Resources and Locks

Key to Lock Modes

NL—Null lock
CR—Concurrent read
CW—Concurrent write
PR—Protected read
PW—Protected write
EX—Exclusive lock

Figure 12-2 Three Lock Queues



ZK-374-81

When a new lock request is made, the lock management services first determine if the resource is currently known (that is, if any other processes have locks on that resource). If the resource is new (that is, no other locks exist on the resource), the lock management services create an entry for the new resource and the requested lock. If the resource is already known, the lock management services determine if any other locks are waiting in either the conversion or waiting queue. If other locks are waiting in either queue, the new lock request is queued at the end of the waiting queue. If both the conversion and waiting queues are empty, the lock management services determine if the new lock is compatible with the other granted locks. If it is compatible, the lock is granted; if the lock request is not compatible, it is placed on the waiting queue. A flag bit can be used to direct the lock management services not to queue a lock request if it cannot be granted immediately.

Lock Management Services

Concepts of Resources and Locks

12.1.6 Lock Conversion Concepts

Lock conversions allow processes to change the level of locks. For example, a process can maintain a low-level lock on a resource until it wants to limit access to the resource. The process can then request a lock conversion.

Lock conversions are specified using a flag bit (see Section 12.3.5) and a lock status block. The lock status block must contain the lock identification of the lock to be converted. If the new lock mode is compatible with the currently granted locks the conversion request is granted immediately. If the new lock mode is incompatible with the existing locks in the granted queue, the request is placed on the conversion queue. The lock retains its old lock mode and does not receive its new lock mode until the request is granted.

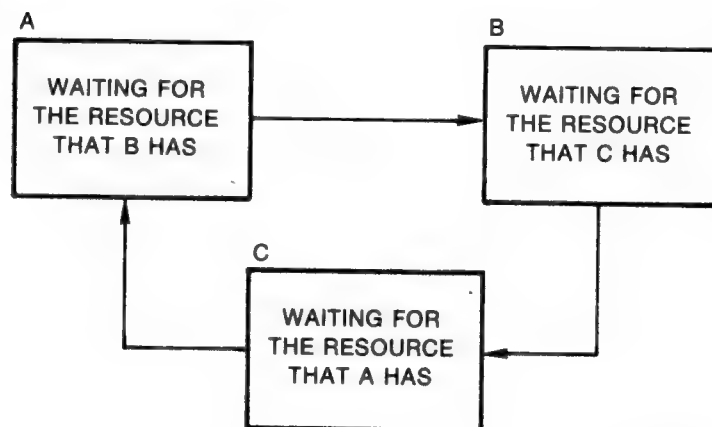
When a lock is dequeued, or converted to a lower lock mode, the lock management services inspect the first conversion request on the conversion queue. If the conversion request is compatible with the locks currently granted, it is granted. Any compatible conversion requests immediately following are also granted. If the conversion queue is empty, the waiting queue is checked. If the first lock request on the waiting queue is compatible with the locks currently granted, it is granted. Any compatible lock requests immediately following are also granted.

12.1.7 Deadlock Detection

A deadlock occurs when any group of locks are waiting for each other in a circular fashion. An example of this situation is shown in Figure 12-3.

In the example, three processes have queued requests for resources that cannot be accessed until the current locks held are dequeued (or converted to a lower lock mode).

Figure 12-3 A Deadlock



ZK-375-81

If the lock management services determine that a deadlock exists, a process is chosen (by the services) to break the deadlock. The chosen process is termed the victim. If the victim has requested a new lock, the lock is not granted; if the victim has requested a lock conversion, the lock is returned to its old lock mode. In either case, the status code SS\$_DEADLOCK is placed in the lock

Lock Management Services

Concepts of Resources and Locks

status block. Note that granted locks are never revoked; only waiting lock requests can receive the status code `SS$_DEADLOCK`.

Note: Programs must not make assumptions regarding which process will be chosen to break a deadlock.

12.2 Queuing Simple Lock Requests

The `$ENQ` system service is used to queue lock requests. When new locks are requested, the lock mode, address of the lock status block, and resource name must be specified in the system service call. The following example illustrates a simple call to `$ENQ`.

EXAMPLE OF A SIMPLE LOCK REQUEST

```
LKSB: .BLKQ 0 ; To contain lock status block
RESOURCE:
.ASCID /STRUCTURE_1/ ; STRUCTURE_1 is the name of
; the resource being locked

$ENQW_8 LKMODE=#LCK$K_PMODE, - ; protected read mode
LKSB=LKSB, -
RESNAM=RESOURCE
```

In the example, the data structure, `STRUCTURE_1`, is accessed by a number of processes. Some processes read the data structure; others write to the structure. Readers must be protected from reading the structure while it is being updated by writers. The reader in the example queues a request for a protected read mode lock. Protected read mode is compatible with itself, so all readers can read the structure at the same time. A writer to the structure uses protected write or exclusive mode locks. Since protected write mode and exclusive mode are not compatible with protected read mode, no writers can write the data structure until the readers have released their locks; and no readers can read the data structure until the writers have released their locks.

Table 12-1 shows the compatibility of lock modes.

12.3 Advanced Locking Techniques

The previous sections discussed locking techniques and concepts useful to all applications. The following sections discuss specialized features of the VAX/VMS lock manager.

12.3.1 Synchronizing Locks

The `$ENQ` system service returns control to the calling program when the lock request is queued. The status code in `R0` indicates whether the request was queued successfully. Once the request is queued, the procedure cannot access the resource until the request is granted. A procedure can use three methods to check that a request has been granted:

- Specify the number of an event flag to be set when the request is granted and wait for the event flag.

Lock Management Services

Advanced Locking Techniques

- Specify the address of an AST routine to be executed when the request is granted.
- Poll the lock status block for a return status code that indicates that the request has been granted.

These methods of synchronization are identical to the synchronization techniques used with the \$QIO system services, described in Section 7.3.

The \$ENQW macro performs synchronization by combining the functions of the \$ENQ system service and the Synchronize (\$SYNCH) system service. The \$ENQW macro has the same arguments as the \$ENQ macro. It queues the lock request, and then places the program in an event flag wait state (LEF) until the lock request is granted.

12.3.2 Notification of Synchronous Completion

The lock management services provide a mechanism that allows processes to determine if a lock request is granted synchronously, that is, if the lock is not placed on the conversion or waiting queue. This feature can be used to improve performance in applications where most locks are granted synchronously (as is normally the case).

If the flag bit LCK\$M_SYNCSTS is set and a lock is granted synchronously, the status code SS\$_SYNCH is returned in R0; no event flag is set and no AST is delivered.

If the request is not completed synchronously, the success code SS\$_NORMAL is returned; event flags or AST routines are handled normally (that is, the event flag is set and the AST is delivered when the lock is granted).

12.3.3 Lock Status Block

The lock status block receives the final completion status and the lock identification, and optionally contains a lock value block (Figure 12-4). When a request is queued, the lock identification is stored in the lock status block even if the lock has not been granted. This allows a procedure to dequeue locks that have not been granted. See Section 12.4 for more information on the Dequeue Lock Request (\$DEQ) system service.

Figure 12-4 The Lock Status Block

reserved	condition value
lock identification	
16 byte lock value block only used when LCK\$M_VALBLK is set	

ZK-376-81

Lock Management Services

Advanced Locking Techniques

The status code is placed in the lock status block only when the lock is granted (or when errors occur in granting the lock).

The uses of the lock value block are described in Section 12.5.1.

12.3.4 Blocking ASTs

In some applications using the lock management services, it is highly desirable for a process to know if it is preventing another process from locking a resource. The lock management services inform processes of this through the use of blocking ASTs. To enable blocking ASTs, the **blkast** argument of the \$ENQ system service must contain the address of a blocking AST service routine. When the lock prevents another lock from being granted, a blocking AST is delivered and the blocking AST service routine is executed. The **astprm** argument is used to pass a parameter to the blocking AST. For more information on ASTs and AST service routines, see Chapter 5. Some uses of blocking ASTs are described in Section 12.5.2.

12.3.5 Lock Conversions

Lock conversions perform three main functions.

- Maintaining a low-level lock and converting it to a higher lock mode when necessary
- Maintaining values stored in a resource lock value block (described in the paragraphs below)
- Improving performance in some applications

A procedure normally needs an exclusive (or protected write) mode lock while writing data. However, it may not be desirable for the procedure to keep the resource exclusively locked all the time, if it is not known at execution time whether writing will be necessary. Maintaining an exclusive or protected write mode lock prevents other processes from accessing the resource. Lock conversions allow a process to request a low-level lock at first and convert the lock to a higher-level lock mode (protected write mode, for example) only when it needs to write data.

Some applications of locks require the use of the lock value block. If a version number or other data is maintained in the lock value block, it is necessary to maintain at least one lock on the resource so that the value block is not lost. In this case processes convert their locks to null locks, rather than dequeuing them when finished accessing the resource.

In order to improve performance in some applications, all resources that might be locked are locked with null locks during initialization. The null locks can be converted to higher-level locks as needed. Usually a conversion request is faster than a new lock request because the necessary data structures have already been built. However, maintaining any lock for the life of a procedure uses system dynamic memory. Therefore, the technique of creating all necessary locks as null locks and converting them as needed improves performance at the expense of increased storage requirements.

Note: If you specify the flag bit **LCK\$M_NOQUEUE** on a lock conversion and the conversion fails, the new blocking AST address and parameter specified in the conversion request replaces the blocking AST address and parameter specified in the previous \$ENQ request.

Advanced Locking Techniques

12.3.5.1 Queuing Lock Conversions

To perform a lock conversion, a procedure calls the \$ENQ system service with the flag bit LCK\$M_CONVERT set. Lock conversions do not use the **resnam**, **parid**, **acmode**, or **prot** arguments. The lock being converted is identified by the lock identification contained in the lock status block. The following example shows a simple lock conversion. Note that the lock must be granted before it can be converted.

EXAMPLE OF A LOCK CONVERSION

```

LKSB:      .BLKQ      0
RESOURCE:
    .ASCID      /STRUCTURE_1/
    .
    .
$ENQW_S LKMODE=#LCK$K_NLMODE, - ; null lock
        LKSB=LKSB, -
        RESNAM=RESOURCE
    .
    .-----
    . <-----| Lock is |
    .          | granted |
    . -----
$ENQW_S LKMODE=#LCK$K_PWMODE, - ; protected write
        LKSB=LKSB, - ; lock id is in LKSB
        FLAGS=#LCK$M_CONVERT ; conversion
    .
    .

```

12.3.6 Parent Locks

When a lock request is queued it is possible to declare a parent lock for the new lock. When a lock has a parent it is called a sublock. To specify a parent lock, the lock identification of the parent lock is passed in the **parid** argument to the \$ENQ system service. A parent lock must be granted before the sublocks belonging to the parent can be granted.

The benefits of specifying parent locks are:

- Low-level locks (concurrent read or concurrent write) can be held at a coarse granularity (files, for example), while higher-level (protected write or exclusive mode) sublocks are held on resources of a finer granularity (such as records or data items).
- Resources names are unique with each parent (parent locks are part of the resource name).

The following paragraphs describe the use of parent locks. Assume a number of processes need to access a database. There are two levels where the database can be locked: the file, and individual records. When updating all the records in a file, it is faster and more efficient to lock the whole file and update the records without additional locking. But when updating selected records, it is better to lock each of the records as it is needed.

Lock Management Services

Advanced Locking Techniques

To use parent locks in this way, all processes request locks on the file. Processes that need to update all records must request protected write or exclusive mode locks on the file. Processes that need to update individual records request concurrent write mode locks on the file, and then use sublocks to lock the individual records in protected write or exclusive mode.

In this way the processes that need to access all records can do so by locking the file, while processes that can share the file lock individual records. A number of processes can share the file-level lock at concurrent write mode, while their sublocks update selected records.

The number of levels of sublocks is limited by the size of the interrupt stack. If the limit is exceeded, the error status `SS$_EXDEPTH` is returned. The size of the interrupt stack is controlled by the `SYSGEN` parameter `INTSTKPAGES`. The default value for `INTSTKPAGES` allows 32 levels of sublocks. For more information on `SYSGEN` and `INTSTKPAGES`, see the *Guide to VAX/VMS System Management and Daily Operations*.

12.3.7 Lock Value Blocks

The lock value block is an optional 16-byte extension of a lock status block. The first time a process associates a lock value block with a particular resource, the lock management services create a resource lock value block for that resource. The resource lock value block is maintained by the lock management services until there are no more locks on the resource.

To associate a lock value block with a resource, the flag bit `LCK$_M_VALBLK` must be set in calls to the `$ENQ` system service. The lock status block `lksb` argument must contain the address of the lock status block for the resource.

When a process sets the flag bit `LCK$_M_VALBLK` in a lock request (or conversion request) and the lock request (or conversion) is granted, the contents of the resource lock value block are written to the process's lock value block.

When a process sets the flag bit `LCK$_M_VALBLK` on a conversion from protected write or exclusive mode to a lower mode, the contents of the process's lock value block are stored in the resource lock value block.

In this manner, processes can pass the value in the lock value block along with the ownership of a resource.

Table 12-2 shows how lock conversions affect the contents of the process's and the resource's lock value block.

Table 12-2 Effect of Lock Conversion on Lock Value Block

Lock Mode at Which Lock is Held	Lock Mode to Which Lock is Converted					
	NL	CR	CW	PR	PW	EX
NL	Return	Return	Return	Return	Return	Return
CR	Neither	Return	Return	Return	Return	Return
CW	Neither	Neither	Return	Return	Return	Return
PR	Neither	Neither	Neither	Return	Return	Return
PW	Write	Write	Write	Write	Write	Return
EX	Write	Write	Write	Write	Write	Write

Lock Management Services

Advanced Locking Techniques

Key to Lock Modes

NL—Null lock
CR—Concurrent read
CW—Concurrent write
PR—Protected read
PW—Protected write
EX—Exclusive lock

Key to Effects

Return—The contents of the resource lock value block are returned to the process's lock value block.

Neither—The process's lock value block is not written; the resource lock value block is not returned.

Write—The contents of the process's lock value block are written to the resource lock value block.

Note that when protected write or exclusive mode locks are dequeued, using the Dequeue Lock Request (\$DEQ) system service, and the address of a lock value block is specified in the **valblk** argument, the contents of that lock value block are written to the resource lock value block.

12.4 Dequeuing Locks

When a process no longer needs a lock on a resource, the lock can be dequeued. The Dequeue Lock Request (\$DEQ) system service is used to dequeue locks. Dequeuing locks simply means that the specified lock request is removed from the queue that it is in. Locks are dequeued from any queue: granted, waiting, or conversion. When the last lock on a resource is dequeued, the lock management services delete the name of the resource from its data structures.

The four arguments to the \$DEQ macro (**lkid**, **valblk**, **acmode** and **flags**) are optional. The **lkid** argument allows the process to specify a particular lock to be dequeued, using the lock identification returned in the lock status block.

The **valblk** argument contains the address of a 16-byte value lock block. If the lock being dequeued is in protected write or exclusive mode, the contents of the value block are stored in the value block associated with the resource. If the lock being dequeued is in any other mode, the value block is not used. The lock value block can only be used if a particular lock is being dequeued.

There are three flags available: **LCK\$M_DEQALL**, **LCK\$M_CANCEL** and **LCK\$M_INVVALBLK**.

LCK\$M_DEQALL indicates that all locks of the access mode specified with the **acmode** argument and less privileged access modes are to be dequeued. The access mode is maximized with the access mode of the caller. If the flag **LCK\$M_DEQALL** is specified, then the **lkid** argument must be zero (or not specified).

When **LCK\$M_CANCEL** is specified, \$DEQ attempts to cancel a lock conversion request that was queued by \$ENQ. This attempt can only succeed if the lock request has not yet been granted, in which case, the request is in the conversion queue. The **LCK\$M_CANCEL** flag is ignored if the **LCK\$M_DEQALL** flag is specified. For more information on the **LCK\$M_CANCEL** flag see the description of the \$DEQ service in Part II of this manual.

Lock Management Services

Dequeuing Locks

When LCK\$M_INVVALBLK is specified, \$DEQ marks the lock value block, which is maintained for the resource in the lock database, as invalid. See the descriptions of \$DEQ and \$ENQ in Part II of this manual for more information on the LCK\$M_INVVALBLK flag.

EXAMPLE OF DEQUEUEING LOCKS

```
LKSB: .QUAD 0
RESOURCE: ; resource is
          .ASCID /STRUCTURE_1/ ; STRUCTURE_1
.
$ENQ_S LKMODE=#LCK$K_CRMODE, - ; concurrent read mode
       LKSB=LKSB, -
       RESNAM=RESOURCE, -
       ASTADR=READ_UPDATES
.
$DEQ_S LKID=LKSB+4 ;LKSB+4 contains the lock id
```

User mode locks are automatically dequeued when the image exits.

12.5 Local Buffer Caching with the Lock Management Services

The lock management services provide methods for applications to perform local buffer caching (also called distributed buffer management). Local buffer caching allows a number of processes to maintain copies of data (disk blocks, for example) in buffers local to each process and be notified when the buffers contain invalid data due to modifications by another process. In applications where modifications are infrequent, substantial I/O may be saved by maintaining local copies of buffers—hence the names local buffer caching or distributed buffer management. Either the lock value block or blocking ASTs (or both) can be used to perform buffer caching.

12.5.1 Using the Lock Value Block

To support local buffer caching using the lock value block, each process maintaining a cache of buffers maintains a null mode lock on a resource that represents the current contents of each buffer. (For this discussion assume that the buffers contain disk blocks.) The value block associated with each resource is used to contain a disk block "version number." The first time a lock is obtained on a particular disk block, the current version number of that disk block is returned in the process's lock value block. If the contents of the buffer are cached, this version number is saved along with the buffer. To reuse the contents of the buffer, the null lock must be converted to protected read mode or exclusive mode, depending on whether the buffer is to be read or written. This conversion returns the latest version number of the disk block. The version number of the disk block is compared with the saved version number. If they are equal, the cached copy is valid. If they are not equal, a fresh copy of the disk block must be read from disk.

Lock Management Services

Local Buffer Caching with the Lock Management Services

Whenever a procedure modifies a buffer, it writes the modified buffer to disk and then increments the version number prior to converting the corresponding lock to null mode. In this way, the next process that attempts to use its local copy of the same buffer will find a version number mismatch and must read the latest copy from disk, rather than use its cached (now invalid) buffer.

12.5.2 Using Blocking ASTs

Blocking ASTs are used to notify processes with granted locks that another process with an incompatible lock mode has been queued to access the same resource.

Blocking ASTs can be used to support local buffer caching in two ways. One technique involves deferred buffer writes; the other technique is an alternate method of local buffer caching without using value blocks.

12.5.2.1 Deferring Buffer Writes

When local buffer caching is being performed, a modified buffer must be written to disk before the exclusive mode lock can be released. If a large number of modifications are expected (particularly over a short period of time), disk I/O can be reduced by maintaining the exclusive mode lock for the entire time that the modifications are being made, and writing the buffer once. However, this prevents other processes from using the same disk block during this interval. This can be avoided if the process holding the exclusive mode lock has a blocking AST. The process will be notified (by the AST) if another process needs to use the same disk block. The holder of the exclusive mode lock can then write the buffer to disk and convert its lock to null mode (thereby allowing the other process to access the disk block). However, if no other process needs the same disk block, the first process can modify it many times, but only write it once.

12.5.2.2 Buffer Caching

To perform local buffer caching using blocking ASTs, processes do not convert their locks to null mode from protected read or exclusive mode when finished with the buffer. Instead, they receive blocking ASTs whenever another process attempts to lock the same resource in an incompatible mode. With this technique, processes are notified that their cached buffers are invalid as soon as a writer needs the buffer, rather than the next time the process tries to use the buffer.

12.5.3 Choosing a Buffer Caching Technique

The choice between using version number or blocking ASTs to perform local buffer caching depends on the characteristics of the application. An application that uses version numbers performs more lock conversions, while one that uses blocking ASTs delivers more ASTs. Note that both techniques are compatible with each other; some processes can use one technique and other processes can use the other at the same time. Generally speaking, blocking ASTs are preferred in a low-contention environment, while version numbers are preferred in a high-contention environment. It is even possible to invent "combined" or "adaptive" strategies.

In a "combined" strategy, the applications use specific techniques. If a process is expected to reuse the contents of a buffer in a short amount of time, blocking ASTs are used; if there is no reason to expect a quick reuse, version numbers are used.

Lock Management Services

Local Buffer Caching with the Lock Management Services

In an "adaptive" strategy, an application makes evaluations on the rate of blocking ASTs and conversions. If blocking ASTs arrive frequently, the application changes to using version numbers; if many conversions take place and the same cached copy remains valid, the application changes to using blocking ASTs.

For example, consider the case where one process continually displays the state of a database, while another occasionally updates it. If version numbers are used, the displaying process must always check to see that its copy of the database is valid (by performing a lock conversion); if blocking ASTs are used, the display process is informed every time the database is updated. On the other hand, if updates occur frequently, the use of version numbers is preferable to continually delivering blocking ASTs.

12.6 Example of Using Lock Management Services

```
! define lock modes
INCLUDE '($LCKDEF)'

! define lock status block
INTEGER*2 LOCK_STATUS,
2      NULL
INTEGER LOCK_ID
COMMON /LOCK_BLOCK/ LOCK_STATUS,
2      NULL,
2      LOCK_ID

! request a null lock
STATUS = SYS$ENQW (,
2      %VAL(LCK$K_NLMODE),
2      LOCK_STATUS,
2      'TERMINAL',
2      .....)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
IF (.NOT. LOCK_STATUS) CALL LIB$SIGNAL (%VAL(LOCK_STATUS))

! convert the lock to an exclusive lock
STATUS = SYS$ENQW (,
2      %VAL(LCK$K_EXMODE),
2      LOCK_STATUS,
2      %VAL(LCK$M_CONVERT),
2      'TERMINAL',
2      .....)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
IF (.NOT. LOCK_STATUS) CALL LIB$SIGNAL (%VAL(LOCK_STATUS))
```

The program segment above requests a null lock for the resource named TERMINAL. After the lock is granted, the program requests that the lock be converted to an exclusive lock. Note that after SYS\$ENQW returns, the program checks both the status of the system service and the lock status block to ensure that the request completed successfully.

Lock Management Services

Example of Using Lock Management Services

To share a terminal between a parent process and a subprocess, each process requests a null lock on a shared resource name. Then, each time one of the processes wants to perform terminal I/O, it requests an exclusive lock, performs the I/O, and requests a null lock.

Since the lock manager is only effective between cooperating programs, the program that created the subprocess should not exit until the subprocess has exited. To ensure that the parent does not exit before the subprocess, specify an event flag to be set when the subprocess exits (the **num** argument of **LIB\$SPAWN**). Before exiting from the parent program, use **SYS\$WAITFR** to ensure that that event flag has been set. (You can suppress the logout message from the subprocess by using the **SYS\$DELPRC** system service to delete the subprocess instead of allowing the subprocess to exit.)

After the parent process exits, a created process cannot synchronize access to the terminal and should use the **SYS\$BRKTHRU** system service to write to the terminal.

13 Programming Examples

This section presents three VAX MACRO programs: ORION, CYGNUS, and LYRA. These programs do not perform any useful work; they are intended only to illustrate how to call various system services.

Each program is preceded by an introduction identifying the services it uses and the main functions it performs. In addition, the programs themselves contain many comments related to specific data definitions and portions of code.

13.1 Orion Program Example

The program ORION uses the following system services:

\$ASSIGN	Assign I/O Channel
\$QIOW	Form of Queue I/O Request and Synchronize
\$NUMTIM	Convert Binary Time to Numeric Time
\$BINTIM	Convert ASCII String to Binary Time
\$SETIMR	Set Timer
\$WAITFR	Wait for Single Event Flag
\$READEF	Read Event Flags
\$SETPRN	Set Process Name

This sample program illustrates the following:

- 1** Assigning an I/O channel to a terminal and writing messages to the terminal. The device name is specified by the logical name **TERMINAL**. Before ORION is run, the logical name must be assigned an equivalence device name.
- 2** Using the **\$NUMTIM** system service to determine whether the current time is before or after noon. A call to **\$SETIMR** is made conditionally if the time is prior to noon.
- 3** Obtaining a delta time value in the system format to use as input to the Set Timer (**\$SETIMR**) system service.
- 4** Calling the Set Timer system service.
 - a** Event flag—The **\$SETIMR** call is followed by a wait for the specified event flag. When the timer expires, the program calls **\$READEF** and displays the current status of the event flag cluster.
 - b** AST routine—One AST routine is for a delta time interval. The other (conditional) is for an absolute time. In either case, the program continues execution and will be interrupted when the timer requests are processed.

Programming Examples

Orion Program Example

- 5 An example of terminal input. The program prompts for a character string to be used as the process name of the current process. Then it uses this name as input to the \$SETPRN system service.

```
.TITLE ORION SYSTEM SERVICES TEST
.IDENT /01/

; Macro library calls
$IODEF                      ; Define I/O function codes
$SSDEF                      ; Define system status values
$READDEF                   ; Define offsets for $READF

; Local macro defined in private macro library
; MESSAGE Output messages formatted by FAO
.MACRO MESSAGE
$QIOW_S CHAN=TTCHAN, -
        FUNC=$IO$_WRITEVBLK, -
        P1=FAOBUF, -
        P2=FAOLEN, -
        P4=$32
        BSBW ERROR
.ENDM MESSAGE

; Read-only data program section
.PSECT RODATA,NOWRT,NOEXE

; Local Read/Write Data
TTNAME: .ASCID /TERMINAL/      ; Terminal logical name

; FAO control strings and data for timer (AST and event flag) tests
ASCNOON:
        .ASCID /-- 12:00:00.00/ ; Noon in ASCII format
TENSEC: .ASCID /0 00:00:10/     ; Ten seconds delta time in ASCII format
DISPLAYFN:
        .ASCID /CLUSTER 2 CONTENTS: !XL/
TIMSTR: .ASCID                  ; Display message after event flag wait
        .ASCID "!/TIMER ENTRY PROCESSED; CLUSTER 2 = !XL"
NOONMSG: .ASCID                 ; Display message at noon
        .ASCID /I'M YOUR TIME AST ROUTINE; IT'S NOON.../
SECMGDESC:
        .ASCID                 ; Display message from AST routine
        .ASCID "!/TIME AST ROUTINE; DELTA TIME !%T"
TWENTY: .LONG -10*1000*1000+20,-1 ; 20 seconds delta time

; Announcement messages
FAOSTR: .ASCID                 ; Master control string
        .ASCID "!/ORION: !AC " ; Name, message

; Announcement messages and lengths for outputting
HELLO: .ASCII /HELLO...MY NAME IS ORION.../
HELLOLEN:
        .LONG HELLOLEN-HELLO
;
TIMERMSG: .ASCII /BEGINNING TIMER TESTS.../
TIMERLEN:
        .LONG TIMERLEN-TIMERMSG
;
EFNWAITMSG:
        .ASCII /TIMER SET; WAIT TEN SECONDS/
EFNWAITLEN:
        .LONG EFNWAITLEN-EFNWAITMSG
;
ASTWAITMSG:
        .ASCII /TIMER SET; AST IN 20 SECONDS/
ASTWAITLEN:
        .LONG ASTWAITLEN-ASTWAITMSG
```

Programming Examples

Orion Program Example

```
; Prompt for terminal input
PROMPT: .ASCII /ENTER 1-16 CHARACTER NAME FOR PROCESS:/
PROMPTLEN:
        .LONG  PROMPTLEN-PROMPT

; Error message control strings
; ERRSTR  formats error message if a system service fails
; IOERRSTR formats error message if I/O fails
; BADASTSTR formats error message if error in AST routine
ERRSTR:
        .ASCID  "!/SYSTEM SERVICE ERROR AT APP. !XL RO=!XL"
IOERRSTR:
        .ASCID  "!/I/O ERROR; IOSB !XW"
BADASTSTR:
        .ASCID  /BAD AST PARAMETER !UL/
WAKEUP: .ASCII  /AWAKENED.../
WAKEUPLN:
        .LONG  WAKEUPLN-WAKEUP

; Read/write data
        .PSECT  RWDATA,RD,WRT,NOEXE

; FAO control string and buffer for all announcement messages
FAODESC:
        .LONG  80                                ; Descriptor for FAO output buffer
        .ADDRESS -
                FAOBUF                            ; Address of buffer
FAOBUF:  .BLKB  80                                ; FAO buffer
FAOLEN:  .WORD  0                                ; Length of final string, always
        .WORD  0                                ; Need longword for $QIOW

; Buffer to format messages from AST routine; a separate output buffer
; ensures that if the AST is delivered while another message is being
; written into the FAO output buffer, no data or message will be lost.
FASTDESC:
        .LONG  80                                ; Descriptor for FAO output buffer
        .ADDRESS -
                FASTBUF                            ; Address of buffer
FASTBUF:  .BLKB  80                                ; FAO buffer
FASTLEN:  .WORD  0                                ; Length of final string, always
        .WORD  0                                ; Need longword for $QIOW

; Receive channel number assigned to terminal and I/O status here
TTCHAN:  .BLKW  1                                ; Terminal channel
TTIOSB:  .BLKW  1                                ; IOSB for terminal input
        .BLKW  1                                ; Return status
TTLEN:   .BLKW  1                                ; Length of I/O
        .BLKL  1                                ; Device char

; Argument list for $NAME_G form of a system service call
READLST:
        $READEF EFN=32, -
                STATE=EFNTEST

; Buffer to obtain numeric values of components of time. Since
; the only field of interest is the hours field, the remaining
; fields in the buffer are not formatted.
TIMES:   .BLKW  3                                ; Year, month, day
HOURS:   .BLKW  1                                ; Current time in hours
        .BLKW  3                                ; Remainder of buffer
```


Programming Examples

Orion Program Example

```

; Buffer for terminal input (will create input descriptor for
; $SETPRN system service)
NAMEDESC:
    .LONG 15 ; Descriptor setup
    .ADDRESS - ; Initial size of buffer
    NAME ; Address of buffer
NAME: .BLKB 15 ; Name string here

; Fields for timer tests
NOON: .BLKQ 1 ; Will contain 12:00 in system format
TEN: .BLKQ 1 ; Will contain 10 second delta time
EFNTEST:
    .LONG 0 ; Receive status of event flags
EFNTEST2:
    .LONG 0 ; Status after timer test

; Longword to save PC on entry to error handling subroutine
SAVEPC: .BLKL 1

; Code begins here.
.PSECT TIMER, EXE, NOWRT
.ENTRY ORION, "M<R2,R3,R4,R5,R6>" ; Entry mask

; Assign an I/O channel to the device specified by the logical name
; TERMINAL and issue a message indicating we're off and running.
; Do not perform normal error checking here: instead, let the
; command interpreter issue a message based on the status in R0
; if the channel assignment fails.
SETUP:
    $ASSIGN_S -
        DEVNAM=TTNAME, -
        CHAN=TTCHAN
    BLBS R0, 10$ ; All okay, continue
    RET ; Otherwise exit with status in R0
;
10$: $QIOW_S CHAN=TTCHAN, -
    FUNC=$IO$_WRITEVBLK, -
    P1=HELLO, -
    P2=HELLOLEN, -
    P4=$32
    BSBW ERROR

; Call Read Event Flags to get status of event flags before beginning
; tests and use FAO to output the contents of local event flag cluster 2
$READDEF_G -
    READLST
    BSBW ERROR
$FAO_S CTRSTR=DISPLAYFN, -
    OUTBUF=FAODESC, -
    OUTLEN=FAOLEN, -
    P1=EFNTEST
    BSBW ERROR
    MESSAGE ; Use MESSAGE MACRO

; Announce start of timer tests
TIMETEST:
    $QIOW_S CHAN=TTCHAN, -
    FUNC=$IO$_WRITEVBLK, -
    P1=TIMERMSG, -
    P2=TIMERLEN, -
    P4=$32
    BSBW ERROR

```

Programming Examples

Orion Program Example

```

; Call $NUMTIM to find out if it is currently AM or PM. If
; the program is being run in the AM (any time), we'll call
; $SETIMR to notify us via an AST when the time rolls over
; to afternoon. If it's already PM, skip this setting of
; the timer.
    $NUMTIM_S -
        TIMBUF=TIMES
    BSBW    ERROR
    CNPW    HOURS,#12          ; Before or after noon?
    BGEQ    10$               ; After or noon, skip setting timer

; Fall through here: format ASCII string representing 12 noon
; into system quadword time format and call $SETIMR with
; the address of AST service routine to handle timer requests.
    $BINTIM_S -                ; Get binary noon time
        TIMBUF=ASCNOON, -
        TIMADR=NOON
    BSBW    ERROR              ; Error check
    $SETIMR_S -
        DAYTIM=NOON, -
        ASTADR=TIMEAST, -
        REQIDT=#12
    BSBW    ERROR              ; Error check

; Now, get a delta time of 10 seconds formatted into a quadword
10$: $BINTIM_S -                ; Get binary delta time
        TIMBUF=TENSEC, -
        TIMADR=TEN
    BSBW    ERROR              ; Error check
    $SETIMR_S -                ; Set timer (ten seconds)
        EFN=#33, -
        DAYTIM=TEN
    BSBW    ERROR              ; Error check

; Announce wait for event flag and wait; then read the
; event flag cluster and output its contents
    $QIOW_S CHAN=TTCHAN, -
        FUNC=$IO$_WRITEVBLK, -
        P1=EFNWAITMSG, -
        P2=EFNWAITLEN, -
        P4=#32
    $WAITFR_S -
        EFN=#33                ; Now wait
    BSBW    ERROR              ; Error check

; Update argument list for $READEF and then call it with new address
; to write the cluster into. When complete, format a message and
; display the contents of the cluster.
    MOVAL    EFNTEST2,READLST+READEF$_STATE
    $READEF_G -
        READLST
    BSBW    ERROR              ; Error check
    $FAO_S CTRSTR=TIMSTR, -
        OUTLEN=FAOLEN, -
        OUTBUF=FAODESC,-
        P1=EFNTEST2
    BSBW    ERROR              ; Error check
    MESSAGE

```

Programming Examples

Orion Program Example

```

; Announce setting of timer with AST in 20 seconds (using
; alternate method of specifying delta time). Then, set timer
; and continue.
    $QIOW_S CHAN=TTCHAN, -
        FUNC=#IO$_WRITEVBLK, -
        P1=ASTWAITMSG, -
        P2=ASTWAITLEN, -
        P4=#32

    $SETIMR_S -
        DAYTIM=TWENTY, -
        ASTADR=TIMEAST, -
        REQIDT=#20

    BSBW    ERROR                ; Error check

; Issue a prompt for terminal input: request a name for the current
; process and then use the character string entered as the process
; name.
RDNAME:
    $QIOW_S CHAN=TTCHAN, -
        FUNC=#IO$_WRITEVBLK, -
        P1=PROMPT, -
        P2=PROMPTLEN, -
        P4=#32

    BSBW    ERROR                ; Error check

    $QIOW_S CHAN=TTCHAN, -
        FUNC=#IO$_READVBLK, -
        IOSB=TTIOSB, -
        P1=NAME, -
        P2=NAMEDESC

    BSBW    ERROR

    CMPW    TTIOSB,#SS$_NORMAL    ; I/O successful?
    BEQL    10$                  ; Yes, go on

    $FAO_S  CTRSTR=IOERRSTR, -
        OUTLEN=FAOLEN, -
        OUTBUF=FAODESC, -
        P1=TTIOSB

    MESSAGE
    BRW     RDNAME                ; Go try again
10$:      MOVZWL  TLEN,NAMEDESC    ; Update descriptor length
    $SETPRN_S -
        PRCNAM=NAMEDESC          ; Set process name
    BSBW    ERROR

; Hibernate. When ORION is run interactively, the terminal is dormant.
; When the AST for the Set Timer service is delivered, ORION
; will awaken long enough to execute the AST service routine and
; then resume execution.

; If ORION is run in a subprocess, wakeups can be scheduled for
; delta time intervals. Each time it is awakened, ORION displays a
; message and then resumes hibernating.
HIB:      $HIBER_S                ; For now
    $QIOW_S CHAN=TTCHAN, -
        FUNC=#IO$_WRITEVBLK, -
        P1=WAKEUP, -
        P2=WAKEUPLN, -
        P4=#32

    BRB     HIB
    RET

```

Programming Examples

Orion Program Example

; AST routine to handle timer requests

```

.ENTRY TIMEAST, "M<>" ; Entry mask for timer AST routine
CMPL #12,4(AP) ; Is it noon AST?
BEQL 10$ ; Yes, go do it
CMPL #20,4(AP) ; Is it delta time AST?
BEQL 20$ ; Yes, go do that
BRW 30$ ; Neither, issue error message

```

; Format message for noon AST

```

10$: $FAO_S CTRSTR=FAOSTR, -
        OUTBUF=FASTDESC, -
        OUTLEN=FASTLEN, -
        P1=#NOONMSG
BSEW ERROR ; Error check
$QIOW_S CHAN=TTCHAN, -
        FUNC=#IO$_WRITEVBLK, -
        P1=FASTBUF, -
        P2=FASTLEN, -
        P4=#32
BSEW ERROR ; Error check
RET

```

; Format message for 20 second AST

```

20$: $FAO_S CTRSTR=SECMMSGDESC, -
        OUTBUF=FASTDESC, -
        OUTLEN=FASTLEN, -
        P1=#TWENTY
$QIOW_S CHAN=TTCHAN, -
        FUNC=#IO$_WRITEVBLK, -
        P1=FASTBUF, -
        P2=FASTLEN, -
        P4=#32
RET

```

; Format message if spurious AST

```

30$: $FAO_S CTRSTR=BADASTSTR, -
        OUTLEN=FASTLEN, -
        OUTBUF=FASTDESC, -
        P1=4(AP)
$QIOW_S CHAN=TTCHAN, -
        FUNC=#IO$_WRITEVBLK, -
        P1=FASTBUF, -
        P2=FASTLEN, -
        P4=#32
RET

```

; Error-handling routine: checks status code in R0.
; If low bit set, returns to mainline routine. Otherwise,
; displays approximate PC and R0 when system service call
; encounters an error and issues RET that causes image exit.

```

ERROR: BLBC R0,10$ ; If error, branch
        RSB ; Otherwise, continue

```

; Use FAO to format output error message

```

10$: MOVL (SP),SAVEPC
$FAO_S CTRSTR=ERRSTR, -
        OUTLEN=FAOLEN, -
        OUTBUF=FAODESC, -
        P1=SAVEPC, -
        P2=R0
BLBC R0,END
$QIOW_S CHAN=TTCHAN, -
        FUNC=#IO$_WRITEVBLK, -
        P1=FAOBUF, -
        P2=FAOLEN, -
        P4=#32
END: RET
.END ORION

```

Programming Examples

Cygnus Program Example

13.2 Cygnus Program Example

The program CYGNUS uses the following system services:

\$ASSIGN	— Assign I/O Channel
\$DCLEXH	— Declare Exit Handler
\$CREMBX	— Create Mailbox
\$GETDVI	— Get Device/Volume Information
\$CREPRC	— Create Process
\$FAO	— Formatted ASCII Output
\$QIO	— Queue I/O Request
\$CRELNM	— Create Logical Name
\$WAKE	— Wake Process
\$SETSFM	— Set System Service Failure Exception Mode
\$WAITFR	— Wait for Single Event Flag
\$DELLNM	— Delete Logical Name
\$DASSGN	— Deassign I/O Channel

This sample program illustrates the following:

- 1 Assigning a channel to the current output device assigned to the logical name SYS\$OUTPUT.
- 2 Declaring an exit handler to receive control at image exit. The exit handler ensures that the image exits in a graceful manner.
- 3 Creating a mailbox and using the \$GETDVI system service to obtain the unit number.
- 4 Obtaining the logical name translation of SYS\$OUTPUT, and checking for a concealed device name, by using the \$GETDVI system service.
- 5 Creating a subprocess and using the mailbox created as a termination mailbox. When the subprocess terminates, an AST service routine interprets the message.
- 6 Placing names in the group logical name table.
- 7 Waking a hibernating subprocess. The subprocess created by this program places itself in hibernation after getting started. When awakened, it translates the logical names placed in the group logical name table.

```
.IDENT /01/
; System macro definitions required by CYGNUS
$SSDEF          ; Define status codes for returns
$IODEF          ; Define I/O functions codes for $QIO
$MSGDEF         ; Define names for mailbox messages
$PQLDEF         ; Define names for quota list
$ACCDEF         ; Define names for termination message
$DIBDEF         ; Define names for device information buffer
$DVIDEF         ; Define item codes for device information
$LNMDDEF        ; Define item codes for logical names
```

Programming Examples

Cygnus Program Example

```

; Local macros:
; MESSAGE, to output messages formatted by FAO
;
; .MACRO MESSAGE
; $QIOW_S CHAN=TTCHAN, -
; FUNC=#IO$_WRITEVBLK, -
; P1=FAOBUF, -
; P2=FAOLEN
; P4=#32
; BSBW ERROR
; .ENDM MESSAGE

; GRPNAME, to place logical name/equivalence name
; pairs in the group logical name table with $CRELOG and
; do error checking.
;
; .MACRO GRPNAME LOGICAL,EQUAL
; MOVW EQUAL,CREITM
; MOVL EQUAL+4,CREBF
; $CRELNM_S -
; TABNAM=GRPTBL, -
; LOGNAM=LOGICAL, -
; ITMLST=CREITM
; BSBW ERROR
; .ENDM GRPNAME

; Read-only data program section
; .PSECT RODATA,NOWRT,NOEXE

; Descriptor for input logical name
OUTPUT: .ASCID /SYS$OUTPUT/

; Descriptor for group logical name table
GRPTBL: .ASCID /LNM$GROUP/

; Buffers for announcement messages and lengths
HELLO: .ASCID /CYGNUS...HELLO/
HELLOLEN:
; .LONG HELLOLEN-HELLO

;
; BYE: .ASCII /CYGNUS EXIT HANDLER.../
; BYELEN: .LONG BYELEN-BYE

; Control strings for output messages formatted by FAO and associated
; counted ASCII strings to insert in messages
PRCSTR:
; .ASCID /LYRA CREATED, PID !XL/ ; display PID of subprocess
ASTERRSTR:
; .ASCID "!/MAILBOX MESSAGE HAS !AC !XW"
IOERR: .ASCIC 'I/O ERROR' ; I/O error in AST routine
IDERR: .ASCIC /BAD MSG ID/ ; Mailbox message not
; termination message

PIDERRSTR:
; .ASCID "!/SPURIOUS PROCESS ID !XL IN DELETION MAILBOX"
DONESSTR:
; .ASCID "!/LYRA COMPLETED; STATUS !XL TIME !XT"
BADEISTR:
; .ASCID "!/EXIT DUE TO ERROR !XL"

```

Programming Examples

Cygnus Program Example

```

; Item list for $GETDVI to find unit number of mailbox
;
MBX_DVILIST:                                ; Begin $GETDVI item list
.WORD 4                                     ; Maximum of 4 bytes long
.WORD DVI$_UNIT                             ; Item code for unit number
.ADDRESS -
      UNIT_NUMBER                           ; Address of buffer
.LONG 0                                     ; No return length needed
.LONG 0                                     ; End item list
;
; Item list for $GETDVI finding logical name translation of SYS$OUTPUT
;
TERM_DVILIST:                               ; Begin $GETDVI item list
.WORD 64                                    ; Maximum of 64 bytes long
.WORD DVI$_DEVNAM                           ; Item code for device name
.ADDRESS -
      TERM                                  ; Destination of terminal name
.ADDRESS -
      TERM_DESC                             ; Destination of length of string
.LONG 0                                     ; End item list
;
; Descriptor to define name of image for subprocess to execute.
LYRAEXE:
.ASCID /LYRA.EXE/
; Quota list for subprocess: defines minimal quotas required
; for the subprocess to execute and ensures that the creating
; image will have sufficient quotas to continue.
QLIST: .BYTE PQL$_BYTLM                     ; Buffer quota
      .LONG 1024
      .BYTE PQL$_FILLM                       ; Open file quota
      .LONG 3
      .BYTE PQL$_PGFLQUOTA                   ; Paging file quota
      .LONG 256
      .BYTE PQL$_PRCLM                       ; Subprocess quota
      .LONG 1
      .BYTE PQL$_TQELM                       ; Timer queue quota
      .LONG 3
      .BYTE PQL$_LISTEND
; Logical name/equivalence name pairs for group table.
; Note that one of the names in the table is nested.
ORION: .ASCID /ORION/
HUNTER: .ASCID /HUNTER/
PEGASUS: .ASCID /PEGASUS/
HORSE: .ASCID /HORSE/
LYRA: .ASCID /LYRA/
HARP: .ASCID /HARP/
CYG: .ASCID /CYGNUS/
SWAN: .ASCID /SWAN/
DUCK: .ASCID /UGLY DUCKLING/
TALE: .ASCID /FAIRY TALE!/
; Read/write data program section
.PSECT RWDATA,RD,WRT,NOEXE
UNIT_NUMBER:
.LONG 0                                     ; Destination of unit number
TERM_DESC:
.LONG 64                                    ; Maximum of 64 bytes
TERM_ADDRESS:
.ADDRESS -
      TERM
;
CONC_TERM:
.ASCII /_/                                ; 2nd underscore for concealed device
TERM: .BLKB 64                             ; Terminal name is placed here
TTCHAN: .BLKW 1                             ; Channel number of terminal

```

Programming Examples

Cygnus Program Example

```

;CRELNM item list
;This list is filled in for each invocation of the GRPNAME macro
CREITM: .WORD 0 ; Equivalence length
        .WORD LNM$STRING ; Item code
CREBF: .LONG 0 ; Equivalence buffer
        .LONG 0 ; No return length
        .LONG 0 ; List terminator

; Termination control block
EXITBLOCK: ; Exit control block
        .BLKL 1 ; System uses this for pointer
        .ADDRESS -
        EXITRIN ; Address of routine
        .LONG 2 ; Number of arguments for handler
        .ADDRESS -
        STATUS ; Address to store status
ERRPC: .BLKL 1 ; Store PC (if error)
STATUS: .BLKL 1 ; Status code at exit

; Fields used for termination mailbox creation, message buffering
EXCHAN: .BLKW 1 ; Channel number of mailbox
MBXIOSE: ; I/O status block
        .BLKW 1 ; Status of I/O completion
MBLEN: .BLKW 1 ; Length of operation here
MBPID: .BLKL 1 ; PID of process deleted
EXITMSG: ; Buffer for mailbox message
        .BLKB ACC$K_TERMLEN

; Receive PID of subprocess here
LYRAPID:
        .BLKL 1

; Output buffers for strings formatted by FAO
FAODESC: ; Descriptor for output buffer
        .LONG 80 ; 80-character buffer
        .ADDRESS -
        FAOBUF ; Address
FAOBUF: .BLKB 80 ; Buffer
FAOLEN: .BLKW 1 ; Receive length here
        .BLKW 1 ; Need longword for $QIO

; Need separate FAO buffers for use in AST routine to ensure
; that data doesn't get clobbered asynchronously
FASTDESC:
        .LONG 80 ; Length
        .ADDRESS -
        FASTBUF ; Address
FASTBUF: .BLKB 80 ; Buffer
FASTLEN:
        .BLKW 1 ; Get length
        .BLKW 1 ; Need longword for $QIO

; Program code begins here.
        .PSECT CODE,EXE,RD,NOWRT
        .ENTRY CYGNUS,"M<R2,R3,R4,R5,R6,R7,R8,R9,R10,R11>"

; Call $ASSIGN to assign an I/O channel to device assigned to SYS$OUTPUT
; and issue message verifying successful initialization
10$: $ASSIGN_8 -
        DEVNAM=OUTPUT, -
        CHAN=TTCHAN
        BSBW ERROR ; Error check
        $QIO_8 CHAN=TTCHAN, -
        FUNC=$IO$_WRITEVBLK, -
        P1=HELLO, -
        P2=HELLOLEN, -
        P4=$32
        BSBW ERROR

```


Programming Examples

Cygnus Program Example

```
; Declare exit handler to do cleanup operations
$DCLEXH_S -
    DESEBLK=EXITBLOCK
BSBW ERROR

; Create a mailbox for subprocess termination message
;
MAILBOX:
$CREMBX_S -
    CHAN=EXCHAN, -
    MAXMSG=#120, -
    BUFQUO=#240, -
    PROMSK=#0
BSBW ERROR

;
; Use $GETDVI to determine the unit number of the mailbox
;
$GETDVI_S -
    EFN=#2, - ; Specify event flag
    CHAN=EXCHAN, - ; Channel just assigned
    ITMLST=MBX_DVILIST ; List of information
BSBW ERROR

;
$WAITFR_S - ; Wait for synchronous completion
    EFN=#2
BSBW ERROR

; Translate the logical name SYS$OUTPUT, using $GETDVI
;
$GETDVI_S -
    EFN=#2, - ; Specify event flag
    DEVNAM=OUTPUT, - ; Descriptor for SYS$OUTPUT
    ITMLST=TERM_DVILIST ; List of information
BSBW ERROR

;
$WAITFR_S - ; Wait for synchronous completion
    EFN=#2
BSBW ERROR

;
CMPL RO,$SS$_CONCEALED ; Was the device concealed?
BNEQ PROCESS ; No, branch
INCR TERM_DESC ; Yes, add one to length of name...
DECL TERM_ADDRESS ; and change pointer to CONC_TERM

;
; Create the subprocess. The logical name SYS$OUTPUT will be
; equated to the same device as SYS$OUTPUT of the creating process.
; The MBXUNT argument specifies the name of the mailbox just
; created; the mailbox will receive a message when LYRA exits.
PROCESS:
$CREPRC_S -
    IMAGE=LYRAEXE, -
    PIDADR=LYRAPID, -
    MBXUNT=UNIT_NUMBER, -
    OUTPUT=TERM_DESC, -
    QUOTA=QLIST
BSBW ERROR

; If okay, format an output message showing the process id.
$FAO_S CTRSTR=PRCSTR, -
    OUTLEN=FAOLEN, -
    OUTBUF=FAODESC, -
    P1=LYRAPID
BSBW ERROR
$QIOW_S CHAN=TTCHAN, -
    FUNC=#IO$_WRITEVBLK, -
    P1=FAOBUF, -
    P2=FAOLEN, -
    P4=#32
BSBW ERROR
```

Programming Examples

Cygnus Program Example

```
; Queue an I/O request to the mailbox with an AST
; to receive notification when LYRA completes.
```

```
    $QIO_S  EFN=#4, -
            CHAN=EXCHAN, -
            FUNC=$IO$_READVBLK,-
            ASTADR=EXITAST, -
            IOSB=MBXIOSB,-
            P1=EXITMSG, -
            P2=$ACC$_K_TERMLEN
    BSBW    ERROR
```

```
; Place names in the group logical name table using the macro GRPNAME.
; It will be LYRA's task, when awakened, to translate these
; names and display the results at the terminal.
; Note that translation of the name CYGNUS will require
; iterative translation.
```

```
PUT_NAMES:
```

```
    GRPNAME ORION,HUNTER
    GRPNAME PEGASUS,HORSE
    GRPNAME LYRA,HARP
    GRPNAME CYG,SWAN
    GRPNAME SWAN,DUCK
    GRPNAME DUCK,TALE
```

```
; After placing names in the table, wake LYRA, which has been hibernating,
; to perform the logical name translation.
```

```
    $WAKE_S PIDADR=LYRAPID
    BSBW    ERROR
    RET                                           ; All finished
```

```
; AST service routine to read the termination mailbox.
; In this example, only one message is actually expected in the mailbox
; but the program performs all the following checks:
```

- ; 1. That the I/O completed successfully.
- ; 2. That the message in the mailbox is a process termination message.
- ; 3. That the process being deleted is the subprocess created.

```
; This service routine enables system service failure exception
; mode as an error-handling device: if a system service
; call fails, an exception condition will occur. CYGNUS
; does not declare a condition handler, so the image
; will be forced to terminate, and the system will display
; pertinent information about the exception condition.
```

```
    .ENTRY EXITAST,"M<R2,R3,R4,R5,R6,R7,R8,R9,R10,R11>
    $SETSF_M_S -
            ENBFLG=#1                               ; Enable SSFAIL exceptions
```

```
; Check IOSB to ensure that I/O completed successfully
```

```
    CMPW    MBXIOSB,$SS$_NORMAL                    ; Check that I/O was successful
    BEQL    20$                                     ; Okay, go on
    $FAO_S  CTRSTR=ASTERSTR,-                        ; Otherwise, format error msg
            OUTLEN=FASTLEN, -
            OUTBUF=FASTDESC-
            P1=$IOERR, -                             ; I/O error
            P2=MBXIOSB                               ; Display IOSB
    $QIOW_S CHAN=TTCHAN, -
            FUNC=$IO$_WRITEVBLK, -
            P1=FASTBUF, -
            P2=FASTLEN, -
            P4=#32
    BRW     50$                                     ; Return
```

Programming Examples

Cygnus Program Example

```
; Check message type field in mailbox message to ensure that the message
; is a process termination message.
20$:  CMPW    EXITMSG+ACC$W_MSGTYP,$MSG$_DELPROC      ; Check message type
      BEQL    30$                                     ; Okay, go on
      $FAO_S  CTRSTR=ASTERRSTR,-                      ; Otherwise, format error message
              OUTLEN=FASTLEN, -
              OUTBUF=FASTDESC,-
              P1=$IDERR, -                            ; Invalid PID error
              P2=EXITMSG+ACC$W_MSGTYP ; Print message type code
      $QIOW_S CHAN=TTCHAN, -
              FUNC=$IO$_WRITEVBLK, -
              P1=FASTBUF, -
              P2=FASTLEN, -
              P4=$32
      BRW     50$                                     ; Return

; Compare the second longword in the IOSB with the PID returned
; by $CREPRC to ensure that the termination message is for LYRA.
30$:  CMPL    LYRAPID,MBPID                          ; LYRA deletion?
      BNEQ    35$                                     ; Yes, go on
      BRW     40$

35$:  $FAO_S  CTRSTR=PIDERRSTR,-                      ; Otherwise, format error message
              OUTLEN=FASTLEN, -
              OUTBUF=FASTDESC,-
              P1=MBPID                                ; Display spurious PID
      $QIOW_S CHAN=TTCHAN, -
              FUNC=$IO$_WRITEVBLK, -
              P1=FASTBUF, -
              P2=FASTLEN, -
              P4=$32
      BRW     50$                                     ; Return

; Format an output message indicating LYRA's final exit status
; and the time of day at which LYRA terminated.
40$:  $FAO_S  CTRSTR=DONESTR, -                      ; Format message telling
              OUTLEN=FASTLEN, -                      ; of LYRA's demise
              OUTBUF=FASTDESC,-
              P1=EXITMSG+ACC$L_FINALSTS, -           ; Get status code
              P2=$EXITMSG+ACC$Q_TERMTIME             ; and time of deletion
      $QIOW_S CHAN=TTCHAN, -
              FUNC=$IO$_WRITEVBLK, -
              P1=FASTBUF, -
              P2=FASTLEN, -
              P4=$32
50$:  $SETSFM_S -
      ENBFLG=$0                                     ; Disable exceptions
      RET                                           ; Return
```

Programming Examples

Cygnus Program Example

```
; This is the exit handler for CYGNUS. It receives control
; when CYGNUS exits, either normally, or as a result of
; an error condition.
        .ENTRY EXITRTN,"M<>"          ; Entry mask
        $QIOW_S CHAN=TTCHAN, -
        FUNC=#IO$_WRITEVBLK, -
        P1=BYE, -
        P2=BYELEN, -
        P4=#32
        BSBW ERROR
        BLBS STATUS,20$                ; Normal exit, continue
; If error, format error message using argument list in
; exit control block
10$:    $FAO_S CTRSTR=BADEXSTR, -
        OUTLEN=FAOLEN, -
        OUTBUF=FAODESC,-
        P1=STATUS, -
        P2=ERRPC
        BSBW ERROR
        $QIOW_S CHAN=TTCHAN, -
        FUNC=#IO$_WRITEVBLK, -
        P1=FAOBUF, -
        P2=FAOLEN, -
        P4=#32
; Common code for both normal and error exit: wait for subprocess
; to terminate (if it hasn't already), then delete all names
; from the group logical name table.
20$:    $WAITFR_S -
        EFN=#4                        ; Wait for termination message
        BSBW ERROR
30$:    $DELLNM_S -
        TABNAM=GRPTBL                ; Delete all names
        BSBW ERROR
        $DASSGN_S -
        CHAN=EXCHAN                  ; Deassign mailbox channel
        BSBW ERROR
        MOVL STATUS,RO                ; Restore saved status code
        RET                          ; Exit with status
; Common error handling routine. This routine checks the
; status code in RO; if success, returns to main
; program. If there is an error, the PC is placed in the exit
; control block so that exit routine can format and display
; an error message.
ERROR:
        BLBC RO,10$                  ; Check status code
        RSB                          ; Low bit set, go back
10$:    MOVL (SP),ERRPC                ; Store PC
        RET                          ; RET will cause image exit
        .END CYGNUS
```

13.3 Lyra Program Example

The program LYRA uses the following system services:

\$TRNLNM — Translate Logical Name
 \$ASSIGN — Assign I/O Channel
 \$HIBER — Hibernate
 \$FAOL — Formatted ASCII Output with List Parameter

LYRA is the subprocess created by CYGNUS. After assigning a channel to its current output device, LYRA hibernates. When awakened by CYGNUS,

Programming Examples

Lyra Program Example

LYRA translates the logical names placed in the group logical name table by CYGNUS, and displays the results of the translations on the terminal.

When LYRA exits, a termination message is sent to the mailbox specified by CYGNUS.

```
.IDENT /01/

; Macro library call
    $SSDEF                      ; Define system status values
    $LNMDEF                     ; Define logical name item codes

; Local macro
; MESSAGE, to output messages formatted by FAO
    .MACRO MESSAGE
    $QIOW_S CHAN=TTCHAN, -
        FUNC=#IO$_WRITEVBLK, -
        P1=FAOBUF, -
        P2=FAOLEN, -
        P4=#32
    BSEW ERROR
    .ENDM MESSAGE

; Local data program section starts here
    .PSECT RODATA,NOWRT,NOEXE

; Logical name of logical output device
OUTPUT: .ASCID /SYS$OUTPUT/

; Group logical name table
GRPTBL: .ASCID /LNM$GROUP/

; Announcement messages
HELLO: .ASCII /LYRA: INITIALIZING...AND SO TO SLEEP/
HELLOLEN:
    .LONG HELLOLEN-HELLO

;
WAKEMSG:
    .ASCII /LYRA: OKAY, WILL DO LOGICAL NAME TRANSLATION.../
WAKELEN:
    .LONG WAKELEN-WAKEMSG

; FAO control string for logical name output message
LOGNAMSTR:
    .ASCID "!/LYRA: !AS IS A !AS"

; Error message control string
ERRSTR:
    .ASCID "!/LYRA: SYSTEM SERVICE ERROR AT APP. !XL RO=!XL"

; Logical names to be translated
ORIONLOG:
    .ASCID /ORION/
CYGNUSLOG:
    .ASCID /CYGNUS/
LYRALOG:
    .ASCID /LYRA/
PEGASUSLOG:
    .ASCID /PEGASUS/
```

Programming Examples

Lyra Program Example

```
; Read/write data program section starts here
.PSECT RWDATA, RD, WRT, NOEXE

; Item list for $TRNLNM
TRNITM: .WORD 255 ; Buffer length
        .WORD LNM$ _STRING ; Item code
        .LONG 0 ; Buffer address
        .ADDRESS - ; Returned String Length
        .LONG 0 ; List terminator
        .LONG 0 ; List terminator

; Output buffer for all output formatted by FAO
FAOLEN: .WORD 0 ; Length of final string, always
        .WORD 0 ; Need longword for $OUTPUT
FAODESC:
        .LONG 80
        .ADDRESS - ; Address of buffer
        .FAOBUF
FAOBUF: .BLKB 80

; Word to receive channel number of terminal
OUTCHAN:
        .BLKW 1

; Buffers to maintain logical name/equivalence name pairs
; in routine that performs logical name translation
LOGBUFA:
        .LONG 255
        .ADDRESS -
        .BUFA
BUFA: .BLKB 255
LOGBUFB:
        .LONG 255
        .ADDRESS -
        .BUFB
BUFB: .BLKB 255
LOGLEN: .LONG 0 ; Save length of equivalence name

; Parameter list for call to FAOL (used by translate routine)
TLIST:
TLOGNAM:
        .LONG 0 ; Address of logical name descriptor
TEQLNAM:
        .LONG 0 ; Address of equivalence descriptor
SAVER3: .LONG 0 ; Save register contents for switch

; Longword to store the PC when a system service call results in an
; error. LYRA checks the low bit of R0 following each service call.
; If set, LYRA continues; otherwise, it saves the PC and branches
; to an error-handling routine that displays the saved PC and the
; contents of R0.
ERRPC: .LONG 0 ; For address of SSFAIL
```

Programming Examples

Lyra Program Example

```
; Code begins here.
.PSECT CODE,EXE,RD,NOWRT
.ENABL LSB
.ENTRY LYRA,"M<R2,R3,R4,R5,R6>" ; Entry mask
; Assign channel to device referred to by logical name
; SYS$OUTPUT. This name was placed in the logical name
; table by CYGNUS (it is also CYGNUS's logical output device).
20$:  $ASSIGN_S -
      DEVNAM=OUTPUT, -
      CHAN=OUTCHAN
      BLBS  R0,30$
      RET
; Exit with status if ASSIGN fails
30$:  $QIOW_S CHAN=OUTCHAN, -
      FUNC=$IO$_WRITEBLK, -
      P1=HELLO, -
      P2=HELLOLEN, -
      P4=$32
      BLBS  R0,40$
      MOVAL 30$,ERRPC
      BRW   ERROR
40$:  $HIBER_S
      BLBS  R0,50$
      MOVAL 40$,ERRPC
      BRW   ERROR
50$:  $QIOW_S CHAN=OUTCHAN, -
      FUNC=$IO$_WRITEBLK, -
      P1=WAKEMSG, -
      P2=WAKELEN, -
      P4=$32
      BLBS  R0,60$
      MOVAL 50$,ERRPC
      BRW   ERROR
60$:
; When awakened, begin translating logical names. To translate the
; names, place address of a logical name descriptor in R2 and then
; go to the subroutine that performs the translation. Repeat for
; each logical name to translate.
      MOVAL ORIONLOG,R2
      JSB   TRANSLATE
      MOVAL CYGNUSLOG,R2
      JSB   TRANSLATE
      MOVAL LYRALOG,R2
      JSB   TRANSLATE
      MOVAL PEGASUSLOG,R2
      JSB   TRANSLATE
; All finished, return
      RET
```

Programming Examples

Lyra Program Example

```

        .ENABL  LSB
; Subroutine to translate and print logical names:
; On entry to this subroutine,
; R2 = address of logical name to translate
; It uses: R3 to hold address of final result buffer
;          R4 to hold address of intermediate buffer
;
TRANSLATE:
        MOVAL  LOGBUFA,R3          ; Get addresses of buffers
        MOVAL  LOGBUFB,R4

; Initial translation places resultant equivalence name in buffer pointed
; to by R3
10$:    MOVL   4(R3),TRNITM+4
        $TRNLNM_S -
                TABNAM=GRPTBL, -
                LOGNAM=(R2), -
                ITMLST=TRNITM
        BLBS   R0,30$
        MOVAL  10$,ERRPC
        BRW    ERROR

; Place length of equivalence name in first word of descriptor and use this
; descriptor as input for next translation. If SS$NOLOGNAM is returned,
; then there was no nesting of name. If not, update registers to
; provide input and output descriptors for translation and repeat
; translation until SS$NOLOGNAM is returned.
30$:    MOVZWL LOGLEN,(R3)          ; Fix length in buffer
        MOVL   4(R4),TRNITM+4
        $TRNLNM_S -
                TABNAM=GRPTBL, -
                LOGNAM=(R3), -
                ITMLST=TRNITM
        BLBS   R0,40$
        CMPL   R0,$SS$NOLOGNAM
        BEQL   50$
        MOVAL  30$,ERRPC
        BRW    ERROR

40$:    MOVL   R3,SAVER3            ; Switch
        MOVL   R4,R3
        MOVL   SAVER3,R4
        BRB    30$                ; Try again

; Place addresses of logical name and equivalence names in FA0 parameter list
; and call FA0 to format output message, then output the message.
50$:    MOVL   R2,TLOGNAM
        MOVL   R3,TEQLNAM
        $FAOL_S CTRSTR=LOGNAMSTR, -
                OUTLEN=FAOLEN, -
                OUTBUF=FAODESC, -
                PRMLST=TLIST
        BLBS   R0,60$
        MOVAL  50$,ERRPC
        BRW    ERROR

60$:    $QIOW_S CHAN=OUTCHAN, -
                FUNC=#IO$_WRITEVBLK, -
                P1=FAOBUF, -
                P2=FAOLEN, -
                P4=#32
        BLBS   R0,70$
        MOVAL  60$,ERRPC
        BRW    ERROR

70$:    RSB                        ; To main routine

```


Programming Examples

Lyra Program Example

```
; Error-handling routine:  
; This routine uses the saved PC and R0 to format a message describing  
; the conditions under which a call to a system service failed.
```

```
ERROR:
```

```
    $FA0_S CTRSTR=ERRSTR, -  
          OUTBUF=FAODESC, -  
          OUTLEN=FAOLEN, -  
          P1=ERRPC, -  
          P2=R0  
    $QIOW_S CHAN=OUTCHAN, -  
          FUNC=#IO$_WRITEBLK, -  
          P1=FAOBUF, -  
          P2=FAOLEN, -  
          P4=#32  
    RET  
    .END    LYRA
```

PART II System Service Descriptions

\$ADD_HOLDER—Add Holder Record to Rights Database

The Add Holder Record to Rights Database service adds the specified holder record to the target identifier.

FORMAT **SY\$ADD_HOLDER** *id,holder,[attrib]*

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS *id*

VMS Usage: **rights_id**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Target identifier granted to the specified holder when \$ADD_HOLDER completes execution. The **id** argument is a longword containing the binary value of the target identifier.

holder

VMS Usage: **rights_holder**
type: **quadword (unsigned)**
access: **read only**
mechanism: **by reference**

Holder identifier that is granted access to the target identifier when \$ADD_HOLDER completes execution. The **holder** argument is the address of a quadword data structure that consists of a longword containing the holder's UIC identifier followed by a longword containing a value of zero.

attrib

VMS Usage: **mask_longword**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Attributes to be placed in the holder record when the \$ADD_HOLDER completes execution. The **attrib** argument is a longword containing a bit mask specifying the attributes. A holder is granted a specified attribute only if the target identifier has the attribute.

System Service Descriptions

\$ADD_HOLDER

Symbol values are offsets to the bits within the longword. You can also obtain the values as masks with the appropriate bit set using the prefix KGB\$M rather than KGB\$V. The symbols are defined in the system macro library (\$KGBDEF).

Bit	Meaning When Set
KGB\$V_DYNAMIC	Allows the unprivileged holder to add or remove the identifier from the process rights list
KGB\$V_RESOURCE	Allows the holder to charge resources, such as disk blocks, to the identifier

DESCRIPTION The Add Holder Record to Rights Database service registers the specified user as a holder of the specified identifier with the rights database.

Write access to the rights database is required to use this service. If the database is in SYS\$SYSTEM, which is the default, SYSPRV privilege is needed to grant write access to the database.

CONDITION VALUES RETURNED

SS\$_NORMAL	Successful completion.
SS\$_ACCVIO	The holder argument cannot be read by the caller.
SS\$_BADPARAM	The specified attributes contain invalid attribute flags.
SS\$_DUPIDENT	The specified holder already exists in the rights database for this identifier.
SS\$_INSFMEM	Insufficient process dynamic memory is available to open the rights database.
SS\$_IVIDENT	The specified identifier or holder is of invalid format, or the specified identifier and holder are equal.
SS\$_NOSUCHID	The specified identifier does not exist in the rights database, or the specified holder identifier does not exist in the rights database.
RMS\$_PRV	The user does not have write access to the rights database.

Since the rights database is an indexed file accessed with VAX RMS, this service may also return RMS status codes associated with operations on indexed files. Please refer to the *VAX Record Management Services Reference Manual* for descriptions of these status codes.

\$ADD_IDENT—Add Identifier to Rights Database

The Add Identifier to Rights Database service adds the specified identifier to the rights database.

FORMAT **SY\$ADD_IDENT** *name* ,*[id]* ,*[attrib]* ,*[resid]*

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS *name*

VMS Usage: **char-string**
type: **character-coded text string**
access: **read only**
mechanism: **by descriptor—fixed length string descriptor**

Identifier name to be added to the rights database when \$ADD_IDENT completes execution. The **name** argument is the address of the descriptor pointing to the identifier name string.

An identifier name consists of 1 to 31 alphanumeric characters including dollar signs and underscores, containing at least one nonnumeric character. Any lowercase characters specified are automatically converted to uppercase.

id

VMS Usage: **rights_id**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Identifier to be created when \$ADD_IDENT completes execution. The **id** argument is a longword containing the binary value of the identifier to be created.

If omitted, \$ADD_IDENT selects a unique available value from the general identifier space and returns it in **resid**, if it is specified.

attrib

VMS Usage: **mask_longword**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Attributes placed in the identifier's record when \$ADD_IDENT completes execution. The **attrib** argument is a longword containing a bit mask specifying the attributes.

System Service Descriptions

\$ADD_IDENT

Symbol values are offsets to the bits within the longword. You can also obtain the values as masks with the appropriate bit set using the prefix KGB\$M rather than KGB\$V. The symbols are defined in the system macro library (\$KGBDEF).

Bit	Meaning When Set
KGB\$V_DYNAMIC	Allows the unprivileged holder to add or remove the identifier from the process rights list
KGB\$V_RESOURCE	Allows the holder to charge resources, such as disk blocks, to the identifier

resid

VMS Usage: **rights_id**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Identifier value assigned by the system when \$ADD_IDENT completes execution. The **resid** argument is the address of a longword in which the system-assigned identifier value is written.

DESCRIPTION The Add Identifier to Rights Database service adds the specified identifier to the rights database.

Write access to the rights database is required to use this service. If the database is in SYS\$SYSTEM, which is the default, SYSPRV privilege is needed to grant write access to the database.

CONDITION VALUES RETURNED

SS\$_NORMAL	Service is successfully completed.
SS\$_ACCVIO	The name argument cannot be read by the caller, or the resid argument cannot be written by the caller.
SS\$_BADPARAM	The specified attributes contain invalid attribute flags.
SS\$_DUPIIDENT	The specified identifier already exists in the rights database.
SS\$_DUPLNAM	The specified identifier name already exists in the rights database.
SS\$_INSFMEM	Insufficient process dynamic memory is available to open the rights database.
SS\$_IVIDENT	The specified identifier is of invalid format.
RMS\$_PRV	The user does not have write access to the rights database.

Since the rights database is an indexed file accessed with VAX RMS, this service may also return RMS status codes associated with operations on indexed files. Please refer to the *VAX Record Management Services Reference Manual* for descriptions of these status codes.

\$ADJSTK—Adjust Outer Mode Stack Pointer

The Adjust Outer Mode Stack Pointer service modifies the stack pointer for a less privileged access mode. This service is used by the operating system to modify a stack pointer for a less privileged access mode after placing arguments on the stack.

FORMAT **SY\$ADJSTK** [*acmode*] , [*adjust*] , *newadr*

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS ***acmode***

VMS Usage: **access_mode**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Access mode for which the stack pointer is to be adjusted. The **acmode** argument is this longword value. If not specified, a default value of 0 (kernel access mode) is used.

adjust

VMS Usage: **word_signed**
type: **word (signed)**
access: **read only**
mechanism: **by value**

Signed adjustment value to be used to modify the value specified by the **newadr** argument. The **adjust** argument is a signed longword, which is the adjustment value.

Only the low-order word of this argument is used. The value specified by the low-order word is added or subtracted (depending on the sign) from the value specified by the **newadr** argument. The result is loaded into the stack pointer for the specified access mode.

If **adjust** is not specified or is specified as 0, the stack pointer is loaded with the value specified by the **newadr** argument.

See the Description section for additional information about the various combinations of values for **adjust** and **newadr**.

System Service Descriptions

\$ADJSTK

newadr

VMS Usage: **address**
type: **longword (unsigned)**
access: **modify**
mechanism: **by reference**

Value that \$ADJUST is to adjust. The **newadr** argument is the address of this longword value. The value specified by this argument is both read and written by \$ADJSTK. \$ADJSTK reads the value specified and adjusts it by the value of the **adjust** argument (if specified). After this adjustment is made, \$ADJSTK writes the adjusted value back into the longword specified by **newadr** and then loads the stack pointer with the adjusted value.

If the value specified by **newadr** is 0, the current value of the stack pointer is adjusted by the value specified by **adjust**. This new value is then written back into **newadr**, and the stack pointer is modified.

See the Description section for additional information about the various combinations of values for **adjust** and **newadr**.

DESCRIPTION

Combinations of zero and nonzero values for the **adjust** argument and the **newadr** argument provide the following results:

If the adjust argument specifies:	And the value specified by newadr is:	The stack pointer is:
0	0	Not changed
0	An address	Loaded with the address specified
A value	0	Adjusted by the specified value
A value	An address	Loaded with the specified address, adjusted by the specified value

In all cases, the updated stack pointer value is written into the value specified by the **newadr** argument.

CONDITION VALUES RETURNED

SS\$_NORMAL
SS\$_ACCVIO
SS\$_NOPRIV

Service successfully completed.

The value specified by **newadr** or a portion of the new stack segment cannot be written by the caller.

The specified access mode is equal to or more privileged than the calling access mode.

\$ADJWSL—Adjust Working Set Limit

The Adjust Working Set Limit service adjusts a process's current working set limit by the specified number of pages and returns the new value to the caller. The working set limit specifies the maximum number of process pages that may be resident in physical memory.

FORMAT **SY\$ADJWSL** [*pagcnt*],[*wsetlm*]

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS ***pagcnt***

VMS Usage: **longword_signed**
type: **longword (signed)**
access: **read only**
mechanism: **by value**

Signed adjustment value specifying the number of pages to add (if positive) or subtract (if negative) from the current working set limit. The ***pagcnt*** argument is this signed longword value.

If ***pagcnt*** is not specified or is specified as 0, no adjustment is made and the current working set limit is returned in the longword specified by the ***wsetlm*** argument (if this argument is specified).

wsetlm

VMS Usage: **longword_unsigned**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Value of the working set limit, returned by \$ADJWSL. The ***wsetlm*** argument is the address of this longword value. The ***wsetlm*** argument specifies the newly adjusted value if ***pagcnt*** was specified, and it specifies the old, unadjusted value if ***pagcnt*** was not specified.

DESCRIPTION

If a program attempts to adjust the working set limit beyond the system-defined upper and lower limits, no error condition is returned; instead, the working set limit is adjusted to the maximum or minimum size allowed.

The initial value of a process's working set limit is controlled by the working set default (WSDEFAULT) quota. The maximum value to which it may be increased is controlled by the working set extent (WSEXTENT) quota; the minimum value to which it may be decreased is limited by the SYSGEN parameter MINWSCNT.

System Service Descriptions

\$ADJWSL

CONDITION VALUES RETURNED

SS\$_NORMAL
SS\$_ACCVIO

Service successfully completed.

The longword specified by **wsetlm** cannot be written by the caller.

\$ALLOC—Allocate Device

The Allocate Device service allocates a device for exclusive use by a process and its subprocesses. No other process can allocate the device or assign channels to it until the image that called \$ALLOC exits or explicitly deallocates the device with the Deallocate Device (\$DALLOC) service.

FORMAT **SYSS\$ALLOC** *devnam ,[phylen] ,[phybuf] ,[acmode]*
 ,[flags]

RETURNS

VMS Usage: cond_value
type: longword (unsigned)
access: write only
mechanism: by value

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS *devnam*

VMS Usage: **device_name**
type: **character-coded text string**
access: **read only**
mechanism: **by descriptor—fixed length string descriptor**

Device name of the device to be allocated. The **devnam** argument is the address of a character string descriptor pointing to the device name string.

The string may be either a physical device name or a logical name. If it is a logical name, it must translate to a physical device name.

phylen

VMS Usage: word_unsigned
type: word (unsigned)
access: write only
mechanism: by reference

Word into which \$ALLOC writes the length of the device name string for the device it has allocated. The **phylen** is the address of this word.

phybuf

VMS Usage: device_name
type: character-coded text string
access: write only
mechanism: by descriptor—fixed length string descriptor

mechanism: **by descriptor—fixed length string descriptor**
Buffer into which **\$ALLOC** writes the device name string for the device it has allocated. The **phybuf** is the address of a character string descriptor pointing to this buffer.

System Service Descriptions

\$ALLOC

acmode

VMS Usage: **access_mode**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Access mode to be associated with the allocated device. The **acmode** argument is a longword containing the access mode.

The most privileged access mode used is the access mode of the caller. Only equal or more privileged access modes can deallocate the device.

flags

VMS Usage: **mask_longword**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Longword of status flags indicating whether to interpret the **devnam** argument as type of the device to be allocated. Only one flag exists, bit 0. When it is set, the \$ALLOC service allocates the first available device having the type specified in the **devnam** argument.

This feature is available for the following mass storage devices:

RA60	RA80	RA81	RC25
RCF25	RK06	RK07	RL01
RL02	RM03	RM05	RM80
RP04	RP05	RP06	RP07
RX01	RX02	TA78	TA81
TS11	TU16	TU58	TU77
TU78	TU80	TU81	

DESCRIPTION

The calling process must have ALLSPOOL privilege to allocate a spooled device.

When a process calls the Assign I/O Channel (\$ASSIGN) service to assign a channel to a nonshareable, nonspooled device, such as a terminal or line printer, the device is implicitly allocated to the process.

This service can only be used to allocate devices that either exist on the host system or are made available to the host system in a VAXcluster environment.

CONDITION VALUES RETURNED

SS\$_NORMAL	Service successfully completed.
SS\$_BUFFEROVF	Service successfully completed. The physical name returned overflowed the buffer provided, and has been truncated.
SS\$_DEVALRALLOC	Service successfully completed. The device was already allocated to the calling process.
SS\$_ACCVIO	The device name string, string descriptor, or physical name buffer descriptor cannot be read by the caller; or the physical name buffer cannot be written by the caller.

System Service Descriptions

\$ALLOC

SS\$_DEVALLOC

Warning. The device is already allocated to another process, or an attempt to allocate an unmounted shareable device failed because other processes had channels assigned to the device.

SS\$_DEVMOUNT

The specified device is currently mounted and cannot be allocated; or the device is a mailbox.

SS\$_DEVOFFLINE

The specified device is marked offline.

SS\$_IVDEVNAM

No device name string was specified, or the device name string contains invalid characters.

SS\$_IVLOGNAM

The device name string has a length of 0 or has more than 63 characters.

SS\$_IVSTSFLG

Invalid bits are set in the longword of status flags.

SS\$_NODEVAVL

The specified device in a generic search exists but is allocated to another user.

SS\$_NONLOCAL

Warning. The device is on a remote node.

SS\$_NOPRIV

An attempt was made to allocate a spooled device, and the requesting process does not have the required privilege; or the device protection and/or access control list denies access.

SS\$_NOSUCHDEV

Warning. The specified device does not exist in the host system. This error is usually the result of a typographical error.

SS\$_TEMPLATEDEV

An attempt was made to allocate a template device; a template device cannot be allocated.

\$ALLOC can also return any condition value returned by \$ENQ. See the description of \$ENQ for a list of these condition values.

System Service Descriptions

\$ASCEFC

\$ASCEFC—Associate Common Event Flag Cluster

The Associate Common Event Flag Cluster service causes a named common event flag cluster to be associated with a process for the execution of the current image and to be assigned a process-local cluster number for use with other event flag services. If the named cluster does not exist but the process has suitable privilege, the service creates the cluster.

FORMAT

SY\$ASCEFC *efn* ,*name* ,*[prot]* ,*[perm]*

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS *efn*

VMS Usage: **ef_number**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Number of any event flag contained within the desired common event flag cluster. The *efn* argument is a longword value specifying this number.

There are two common event flag clusters: cluster 2 and cluster 3. Cluster 2 contains event flag numbers 64 to 95, and cluster 3 contains event flag numbers 96 to 127. (Clusters 0 and 1 are process-local event flag clusters.)

To associate with common event flag cluster 2, specify any flag number in the cluster (64 to 95); to associate with common event flag cluster 3, specify any event flag number in the cluster (96 to 127).

name

VMS Usage: **ef_cluster_name**
type: **character-coded text string**
access: **read only**
mechanism: **by descriptor—fixed length string descriptor**

Name of the common event flag cluster with which to associate. The *name* argument is the address of a character string descriptor pointing to this name string.

Common event flag clusters are accessible only to processes having the same UIC group number, and each such process must associate with the cluster using the same name (specified in the *name* argument). VAX/VMS implicitly associates the group UIC number with the name, making the name unique to a UIC group.

System Service Descriptions

\$ASCEFC

If the cluster exists in memory shared by multiple processors, the name specified by **name** must contain the name of the shared memory where the cluster resides. In this case, the format for the cluster name is **SHARED-MEMORY-NAME:CLUSTER-NAME**, with a colon separating the shared memory name from the cluster name.

Refer to Section 4.7.1 for more information about common event flag clusters in shared memory.

prot

VMS Usage: **boolean**
type: **byte (unsigned)**
access: **read only**
mechanism: **by value**

Protection specifier that allows or disallows access to the common event flag cluster by processes with the same UIC group number as the creating process. The **prot** argument is a longword value, which is interpreted as Boolean.

The value 0 specifies that any process with the same UIC group number as the creator may access the event flag cluster; this is the default. The value 1 specifies that only processes with the creator's UIC can access the event flag cluster.

perm

VMS Usage: **boolean**
type: **byte (unsigned)**
access: **read only**
mechanism: **by value**

Permanent specifier that marks a common event flag cluster as either permanent or temporary. The **perm** argument is a longword value, which is interpreted as Boolean.

The value 0 specifies that the cluster is temporary; this is the default. The value 1 specifies that the cluster is permanent.

DESCRIPTION The calling process must have the following:

- PRMCEB privilege to create a permanent common event flag cluster
- SHMEM privilege to create a common event flag cluster in memory that is shared by multiple processors

Creation of temporary common event flag clusters uses the process's quota for timer queue entries (TQELM); the creation of a permanent cluster does not affect the quota. The quota is restored to the creator of the cluster when all processes associated with the cluster have disassociated.

Creation of a shared memory common event flag cluster requires CEF PORT QUOTA. This quota is set up by using the SYSGEN command SHARE. This quota is restored when the common event flag cluster is deleted.

When a process associates with a common event flag cluster, that cluster's reference count is increased by 1. The reference count is decreased when a process disassociates from the cluster, whether explicitly with the Disassociate Common Event Flag Cluster (\$DACEFC) service or implicitly at image exit.

Temporary clusters are automatically deleted when their reference count goes to 0; permanent clusters must be explicitly marked for deletion with the Delete Common Event Flag Cluster (\$DLCEFC) service.

System Service Descriptions

\$ASCEFC

Since the \$ASCEFC service automatically creates the common event flag cluster if it does not already exist, cooperating processes need not be concerned with which process executes first to create the cluster. The first process to call \$ASCEFC creates the cluster and the others associate with it regardless of the order in which they call the service.

The initial state for all event flags in a newly created common event flag cluster is 0.

If a process has already associated a cluster number with a named common event flag cluster and then issues another call to \$ASCEFC with the same cluster number, the service disassociates the number from its first assignment before associating it with its second.

CONDITION VALUES RETURNED

SS\$_NORMAL	Service successfully completed.
SS\$_ACCVIO	The cluster name string or string descriptor cannot be read by the caller.
SS\$_EXPORTQUOTA	The process has exceeded the number of event flag clusters with which processes on this port of the multiport (shared) memory can associate.
SS\$_EXQUOTA	The process has exceeded its timer queue entry quota; this quota controls the creation of temporary common event flag clusters.
SS\$_INSFMEM	Insufficient system dynamic memory is available to complete the service.
SS\$_ILLEFC	An illegal event flag number was specified. The cluster number must be in the range of event flags 64 through 127.
SS\$_INTERLOCK	The bit map lock for allocating common event flag clusters from the specified shared memory is locked by another process.
SS\$_IVLOGNAM	The cluster name string has a length of 0 or has more than 15 characters.
SS\$_NOPRIV	The process does not have the privilege to create a permanent cluster; the process does not have the privilege to create a common event flag cluster in memory shared by multiple processors; or the protection applied to an existing cluster by its creator prohibits association.
SS\$_NOSHMBLOCK	No shared memory control block for common event flag clusters is available.
SS\$_SHMNOTCNCT	The shared memory named in the name argument is not known to the system. This error can be caused by a spelling error in the string, an improperly assigned logical name, or the failure to identify the memory as shared at system generation time.

\$ASCTIM—Convert Binary Time to ASCII String

The Convert Binary Time to ASCII String service converts an absolute or delta time from 64-bit system time format to an ASCII string.

FORMAT **SYSSASCTIM** [*timlen*],*timbuf* ,[*timadr*] ,[*cvtfllg*]

RETURNS

VMS Usage: **cond_value**
 type: **longword (unsigned)**
 access: **write only**
 mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS *timlen*

VMS Usage: **word_unsigned**
 type: **word (unsigned)**
 access: **write only**
 mechanism: **by reference**

Length (in bytes) of the ASCII string returned by \$ASCTIM. The *timlen* argument is the address of a word containing this length.

timbuf

VMS Usage: **time_name**
 type: **character-coded text string**
 access: **write only**
 mechanism: **by descriptor—fixed length string descriptor**

Buffer into which \$ASCTIM writes the ASCII string. The *timbuf* argument is the address of a character string descriptor pointing to the buffer.

The buffer length specified in the *timbuf* argument, together with the *cvtfllg* argument, controls what information is returned.

timadr

VMS Usage: **date_time**
 type: **quadword (unsigned)**
 access: **read only**
 mechanism: **by reference**

Time value that \$ASCTIM is to convert. The *timadr* argument is the address of this 64-bit time value. A positive time value represents an absolute time. A negative time value represents a delta time. If a delta time is specified, it must be less than 10,000 days.

If *timadr* is not specified or is specified as 0 (the default), \$ASCTIM returns the current date and time.

System Service Descriptions

\$ASCTIM

cvtfllg

VMS Usage: **longword_unsigned**

type: **longword (unsigned)**

access: **read only**

mechanism: **by value**

Conversion indicator specifying which date and time fields \$ASCTIM should return. The **cvtfllg** argument is a longword value, which is interpreted as Boolean. A value of 1 specifies that \$ASCTIM should return only the hour, minute, second, and hundredth of second fields. A value of 0 (the default) specifies that \$ASCTIM should return the full date and time.

DESCRIPTION

The \$ASCTIM service executes at the access mode of the caller and does not check whether address arguments are accessible before it executes. Therefore, an access violation causes an exception condition if the input time value cannot be read or the output buffer or buffer length cannot be written.

This service does not check the length of the argument list, and therefore cannot return the SS\$_INSFARG (insufficient arguments) condition value.

The ASCII strings returned have the following formats:

Absolute Time: dd-mmm-yyyy hh:mm:ss.cc

Delta Time: dddd hh:mm:ss.cc

Field	Length (Bytes)	Contents	Range of Values
dd	2	Day of month	1-31
-	1	Hyphen	
mmm	3	Month	JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC
-	1	Hyphen	Required syntax
yyyy	4	Year	1858-9999
blank	n	Blank	Required syntax
hh	2	Hour	00-23
:	1	Colon	Required syntax
mm	2	Minutes	00-59
:	1	Colon	Required syntax
ss	2	Seconds	00-59
.	1	Period	Required syntax
cc	2	Hundredths of second	00-99
dddd	4	Number of days (in 24-hr units)	000-9999

Month abbreviations must be uppercase. The hundredths of second field now represents a true fraction; for example, the string .1 represents ten hundredths of a second (one tenth of a second); the string .01 represents one hundredth of a second.

System Service Descriptions

\$ASCTIM

Also, a third digit can be added to the hundredths of second field; this thousandths of second digit is used to round the hundredths of second value. Digits beyond the thousandths of second digits are ignored.

The results of specifying some possible combinations for the values of the **cvtfld** and **timbuf** arguments are shown below:

Time Value	Buffer Length Specified	CVTFLG Argument	Information Returned
Absolute	23	0	Date and time
Absolute	12	0	Date
Absolute	11	1	Time
Delta	16	0	Days and time
Delta	11	1	Time

CONDITION VALUES RETURNED

SS\$_NORMAL
SS\$_BUFFEROVF
SS\$_IVTIME

Service successfully completed.
The buffer length specified in the **timbuf** argument is too small.
The specified delta time is equal to or greater than 10,000 days.

\$ASCTOID—Translate Identifier Name to Identifier

The Translate Identifier Name to Identifier service translates the specified identifier name into its binary identifier value.

FORMAT **SY\$ASCTOID** *name* *,[id]* *,[attrib]*

RETURNS VMS Usage: **cond_value**
 type: **longword (unsigned)**
 access: **write only**
 mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED".

ARGUMENTS *name*

VMS Usage: **char_string**
type: **character-coded text string**
access: **read only**
mechanism: **by descriptor—fixed length string descriptor**
Identifier name translated when \$ASCTOID completes execution. The **name** argument is the address of a descriptor pointing to the identifier name.

id

VMS Usage: **rights_id**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**
Identifier value resulting when \$ASCTOID completes execution. The **id** argument is the address of a longword in which the identifier value is written.

attrib

VMS Usage: **mask_longword**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**
Attributes associated with the identifier returned in **id** when \$ASCTOID completes execution. The **attrib** argument is the address of a longword containing a bitmask specifying the attributes.

Symbol values are offsets to the bits within the longword. You can also obtain the values as masks with the appropriate bit set using the prefix KGB\$M rather than KGB\$V. The symbols are defined in the system macro library (\$KGBDEF).

System Service Descriptions

\$ASCTOID

Bit	Meaning When Set
KGB\$V_DYNAMIC	Allows the unprivileged holder to add or remove the identifier from the process rights list
KGB\$V_RESOURCE	Allows the holder to charge resources, such as disk blocks, to the identifier

DESCRIPTION

The Translate Identifier Name to Identifier converts the specified identifier name to its binary identifier value. Note that when you use wildcards with this service, the records are returned in identifier name order.

CONDITION VALUES RETURNED

SS\$_NORMAL	Successful completion.
SS\$_ACCVIO	The name argument cannot be read by the caller, or the id or attribute arguments cannot be written by the caller.
SS\$_INSFMEM	Insufficient process dynamic memory is available to open the rights database.
SS\$_IVIDENT	The specified identifier is of invalid format.
SS\$_NOSUCHID	The specified identifier name does not exist in the rights database.
RMS\$_PRV	The user does not have read access to the rights database.

Since the rights database is an indexed file accessed with VAX RMS, this service may also return RMS status codes associated with operations on indexed files. Please refer to the *VAX Record Management Services Reference Manual* for descriptions of these status codes.

System Service Descriptions

\$ASSIGN

\$ASSIGN—Assign I/O Channel

The Assign I/O Channel service (1) provides a process with an I/O channel so that input/output operations can be performed on a device or (2) establishes a logical link with a remote node on a network.

FORMAT **SY\$ASSIGN** *devnam ,chan ,[acmode],[mbxnam]*

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS *devnam*

VMS Usage: **device_name**
type: **character-coded text string**
access: **read only**
mechanism: **by descriptor—fixed length string descriptor**

Name of the device to which \$ASSIGN is to assign a channel. The **devnam** argument is the address of a character string descriptor pointing to the device name string.

If the device name contains a double colon (::), the system assigns a channel to the first available network device (NET:) and performs an access function on the network.

chan

VMS Usage: **channel**
type: **word (unsigned)**
access: **write only**
mechanism: **by reference**

Number of the channel that is assigned. The **chan** argument is the address of a word into which \$ASSIGN writes the channel number.

acmode

VMS Usage: **access_mode**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Access mode to be associated with the channel. The **acmode** argument specifies the access mode. The most privileged access mode used is the access mode of the caller. I/O operations on the channel can only be performed from equal and more privileged access modes.

System Service Descriptions

\$ASSIGN

mbxnam

VMS Usage: **device_name**

type: **character-coded text string**

access: **read only**

mechanism: **by descriptor—fixed length string descriptor**

Logical name of the mailbox to be associated with the device. The **mbxnam** argument is the address of a character string descriptor pointing to the logical name string.

If **mbxnam** is specified as 0, no mailbox is associated with the device. This is the default.

The **mbxnam** argument must be specified when performing a nontransparent task-to-task DECnet-VAX operation.

Only the owner of a device can associate a mailbox with the device; the owner of a device is the process that has allocated the device, whether implicitly or explicitly. Only one mailbox can be associated with a device at any one time.

A mailbox cannot be associated with a device if the device has foreign (DEV\$_M_FOR) or shareable (DEV\$_M_SHR) characteristics.

A mailbox is disassociated from a device when the channel that associated it is deassigned.

If a mailbox is associated with a device, the device driver can send status information to the mailbox. For example, if the device is a terminal, this information may indicate dial-up, hang-up, or the reception of unsolicited input; if the device is a network device, it may indicate that the network is connected or perhaps that the line is down.

For details on the nature and format of the information returned to the mailbox, refer to the *VAX/VMS I/O Reference Volume*.

DESCRIPTION

The calling process must have NETMBX privilege to perform network operations.

System dynamic memory is required if the target device is on a remote system.

Channels remain assigned until they are explicitly deassigned with the Deassign I/O Channel (\$DASSGN) service, or, if they are user-mode channels, until the image that assigned the channel exits.

The \$ASSIGN service establishes a path to a device but does not check whether the caller can actually perform input/output operations to the device. Privilege and protection restrictions may be applied by the device drivers.

CONDITION VALUES RETURNED

SS\$_NORMAL

Service successfully completed.

SS\$_REMOTE

Service successfully completed. A logical link is established with the target on a remote node.

SS\$_ABORT

A physical line went down during a network connect operation.

System Service Descriptions

\$ASSIGN

SS\$_ACCVIO	The device or mailbox name string or string descriptor cannot be read by the caller, or the channel number cannot be written by the caller.
SS\$_DEVACTIVE	A mailbox name has been specified, but a mailbox is already associated with the device.
SS\$_DEVALLOC	Warning. The device is allocated to another process.
SS\$_DEVNOTMBX	A logical name has been specified for the associated mailbox, but the logical name refers to a device that is not a mailbox.
SS\$_EXQUOTA	The target of the assignment is on a remote node and the process has insufficient buffer quota to allocate a network control block.
SS\$_INSFMEM	The target of the assignment is on a remote node and there is insufficient system dynamic memory to complete the request.
SS\$_IVDEVNAM	No device name was specified, the logical name translation failed, or the device or mailbox name string contains invalid characters. If the device name is a target on a remote node, this status code indicates that the Network Connect Block has an invalid format.
SS\$_IVLOGNAM	The device or mailbox name string has a length of 0 or has more than 63 characters.
SS\$_NOIOCHAN	No I/O channel is available for assignment.
SS\$_NOLINKS	For network operations, no logical links are available. The maximum number of logical links as set for the NCP executor MAXIMUM LINKS parameter was exceeded.
SS\$_NOPRIV	For network operations, the issuing task does not have the required privilege to perform network operations or to confirm the specified logical link.
SS\$_NOSUCHDEV	Warning. The specified device or mailbox does not exist, or, for DECnet-VAX operations, the network device driver is not loaded (for example, the DECnet-VAX software is not currently running on the local VAX node).
SS\$_NOSUCHNODE	The specified network node is nonexistent or unavailable.
SS\$_REJECT	The network connect was rejected by the network software or by the partner at the remote node, or the target image exited before the connect confirm could be issued.
SS\$_CONNECFAIL	For network operations, the connection to a network object timed out or failed.
SS\$_DEVOFFLINE	For network operations, the physical link is shutting down.
SS\$_FILALRACC	For network operations, a logical link already exists on the channel.

System Service Descriptions

\$ASSIGN

SS\$_INVLOGIN	For network operations, the access control information was found to be invalid at the remote node.
SS\$_LINKEXIT	For network operations, the network partner task was started, but exited before confirming the logical link (that is, \$ASSIGN to SY\$_NET).
SS\$_NOSUCHOBJ	For network operations, the network object number is unknown at the remote node; for a TASK=connect, the named DCL command procedure file cannot be found at the remote node.
SS\$_NOSUCHUSER	For network operations, the remote node could not recognize the login information supplied with the connection request.
SS\$_PROTOCOL	For network operations, a network protocol error occurred, most likely because of network software error.
SS\$_REMRSRC	For network operations, the link could not be established because system resources at the remote node were insufficient.
SS\$_SHUT	For network operations, the local or remote node is no longer accepting connections.
SS\$_THIRDPARTY	For network operations, the logical link connection was terminated by a third party (for example, the System Manager).
SS\$_TOOMUCHDATA	For network operations, the task specified too much optional or interrupt data.
SS\$_UNREACHABLE	For network operations, the remote node is currently unreachable.

System Service Descriptions

\$BINTIM

\$BINTIM—Convert ASCII String to Binary Time

The Convert ASCII String to Binary Time service converts an ASCII string to an absolute or delta time value in the system 64-bit time format suitable for input to the Set Timer (\$SETIMR) or Schedule Wakeup (\$SCHDWK) services.

FORMAT **SY\$BINTIM** *timbuf, timadr*

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS

timbuf

VMS Usage: **time_name**
type: **character-coded text string**
access: **read only**
mechanism: **by descriptor—fixed length string descriptor**

The *timbuf* argument specifies the address of a character string descriptor pointing to the VAX/VMS time string. The VAX/VMS time string specifies the absolute or delta time to be converted by \$BINTIM. The VAX/VMS Data Type Table describes the VAX/VMS time string.

timadr

VMS Usage: **date_time**
type: **quadword (unsigned)**
access: **write only**
mechanism: **by reference**

The *timadr* argument is the address of the VAX/VMS quadword system time, which receives the converted time.

DESCRIPTION

The \$BINTIM service executes at the access mode of the caller and does not check whether address arguments are accessible before it executes. Therefore, an access violation causes an exception condition if the input buffer or buffer descriptor cannot be read or the output buffer cannot be written.

This service does not check the length of the argument list and therefore cannot return the SS\$_INSFARG (insufficient arguments) error status code. If the service does not receive enough arguments (for example, if you omit required commas in the call), errors may result.

The required ASCII input strings have the following format:

Absolute Time: dd-mmm-yyyy hh:mm:ss.cc

System Service Descriptions

\$BINTIM

Delta Time: dddd hh:mm:ss.cc

Field	Length (Bytes)	Contents	Range of Values
dd	2	Day of month	1-31
-	1	Hyphen	Required syntax
mmm	3	Month	JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC
-	1	Hyphen	Required syntax
yyyy	4	Year	1858-9999
blank	n	Blank	Required syntax
hh	2	Hour	00-23
:	1	Colon	Required syntax
mm	2	Minutes	00-59
:	1	Colon	Required syntax
ss	2	Seconds	00-59
.	1	Period	Required syntax
cc	2	Hundredths of second	00-99
dddd	4	Number of days (in 24-hour units)	000-9999

Note that month abbreviations must be uppercase and that the hundredths of second field represents a true fraction. For example, the string .1 represents ten hundredths of a second (one tenth of a second) and the string .01 represents one hundredth of a second. Note also that a third digit can be added to the hundredths of second field; this thousandths of second digit is used to round the hundredths of second value. Digits beyond the thousandths of second digits are ignored.

The following two syntax rules apply to specifying the ASCII input string:

- Any of the date and time fields can be omitted.

For absolute time values, the \$BINTIM service supplies the current system date and time for nonspecified fields. Trailing fields can be truncated. If leading fields are omitted, the punctuation (hyphens, blanks, colons, periods) must be specified. For example, the following string results in an absolute time of 12:00 on the current day.

-- 12:00:00.00

For delta time values, the \$BINTIM service uses a default value of 0 for unspecified hours, minutes, and seconds fields. Trailing fields can be truncated. If leading fields are omitted from the time value, the punctuation (blanks, colons, periods) must be specified. If the number of days in the delta time is 0, a 0 must be specified. For example, the following string results in a delta time of 10 seconds.

0 ::10

Note the space between the 0 in the day field and the two colons.

System Service Descriptions

\$BINTIM

- For both absolute and delta time values, there can be any number of leading blanks, and any number of blanks between fields normally delimited by blanks. However, there can be no embedded blanks within either the date or time fields.

CONDITION VALUES RETURNED

SS\$_NORMAL

Service successfully completed.

SS\$_IVTIME

The syntax of the specified ASCII string is invalid,
or the time component is out of range.

EXAMPLE

Column 1 of the following table lists legal input strings to the \$BINTIM service; column 2 lists the \$BINTIM output of these strings translated through the Convert Binary Time to ASCII String (\$ASCTIM) system service. The current date is assumed to be 14-JUN-1982 04:15:28.00.

Input to \$BINTIM	\$ASCTIM output string
- :50	14-JUN-1982 04:50:28.00
-1983 0:0:0.0	14-JUN-1983 00:00:00.00
9-NOV-1982 12:32:1.1161	9-NOV-1982 12:32:01.12
22-APR-1983 16:35:0.0	22-APR-1983 16:35:00.00
0 ::1	0 00:00:00.10
0 ::.06	0 00:00:00.06
5 3:18:32.068	5 03:18:32:07
20 12:	20 12:00:00.00
0 5	0 05:00:00.00

\$BRKTHRU—Breakthrough

The Breakthrough service sends a message to one or more terminals.

The \$BRKTHRU service completes asynchronously; that is, it returns to the caller after queuing the message request, without waiting for the message to be written to the specified terminal(s).

For synchronous completion, use the Breakthrough and Wait (\$BRKTHRUW) service. The \$BRKTHRUW service is identical to the \$BRKTHRU service in every way except that \$BRKTHRUW returns to the caller after the message is written to the specified terminal(s).

For additional information about system service completion, refer to the Synchronize (\$SYNCH) service and to Section 2.5.

The \$BRKTHRU service supersedes the Broadcast (\$BRDCST) service. New programs should be written using \$BRKTHRU, and old programs using \$BRDCST should be converted to use \$BRKTHRU as convenient.

FORMAT	SYS\$BRKTHRU <i>[efn] ,[msgbuf] ,[sendto] ,[sndtyp] ,[iosb] ,[carcon] ,[flags] ,[reqid] ,[timeout] ,[astadr] ,[astprm]</i>
---------------	---

RETURNS	VMS Usage: cond_value type: longword (unsigned) access: write only mechanism: by value
----------------	---

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS	efn VMS Usage: ef_number type: longword (unsigned) access: read only mechanism: by value
------------------	---

Number of the event flag to be set when the message has been written to the specified terminal(s). The **efn** argument is a longword containing this number.

When the message request is queued, \$BRKTHRU clears the specified event flag (or event flag 0 if **efn** was not specified). Then when the message has been sent, \$BRKTHRU sets the specified event flag (or event flag 0).

System Service Descriptions

\$BRKTHRU

msgbuf

VMS Usage: **char_string**

type: **character-coded text string**

access: **read only**

mechanism: **by descriptor—fixed length string descriptor**

Message text to be sent to the specified terminal(s). The **msgbuf** argument is the address of a descriptor pointing to this message text.

The \$BRKTHRU service permits the message text to be as long as 16,350 bytes; however, both the SYSGEN parameter MAXBUF and the caller's available process space may affect the maximum length of the message text.

sendto

VMS Usage: **char_string**

type: **character-coded text string**

access: **read only**

mechanism: **by descriptor—fixed length string descriptor**

Name of a single device (terminal) or single username to which to send the message. The **sendto** argument is the address of a descriptor pointing to this name.

The **sendto** argument is used in conjunction with the **sndtyp** argument. When **sndtyp** specifies BRK\$_DEVICE or BRK\$_USERNAME, the **sendto** argument is required because it specifies the name of the device or username to which to send the message.

If **sndtyp** is not specified or if it does not specify BRK\$_DEVICE or BRK\$_USERNAME, **sendto** should not be specified; if **sendto** is specified, \$BRKTHRU ignores it.

sndtyp

VMS Usage: **longword_unsigned**

type: **longword (unsigned)**

access: **read only**

mechanism: **by value**

Terminal type to which \$BRKTHRU is to send the message. The **sndtyp** argument is a longword value specifying the terminal type.

Each terminal type has a symbolic name, which is defined by the \$BRKDEF macro. The following list describes each terminal type:

Terminal type	Description
BRK\$_ALLUSERS	When specified, \$BRKTHRU sends the message to all users who are currently logged on the system.
BRK\$_ALLTERMS	When specified, \$BRKTHRU sends the message to all terminals at which users are logged on and all other terminals that are connected to the system except those with the AUTOBAUD characteristic set.
BRK\$_DEVICE	When specified, \$BRKTHRU sends the message to a single terminal; the name of the terminal must be specified using the sendto argument.
BRK\$_USERNAME	When specified, \$BRKTHRU sends the message to a user with a specified username; the username must be specified using the sendto argument.

System Service Descriptions

\$BRKTHRU

iosb

VMS Usage: **io_status_block**
type: **quadword (unsigned)**
access: **write only**
mechanism: **by reference**

I/O status block that is to receive the final completion status. The **iosb** is the address of this quadword block.

When the **iosb** argument is specified, \$BRKTHRU sets the quadword to zero when it queues the message request. Then when the message has been sent to the specified terminal(s), \$BRKTHRU returns four informational items, one item per word, in the quadword I/O status block.

These informational items indicate the status of the messages sent only to terminals and mailboxes on the local VAX node; these items do not include the status of messages sent to terminals and mailboxes on other VAX nodes in a VAXcluster.

The following lists, in order, each word of the quadword block and the informational item it contains:

Word	Informational Item
1	A condition value describing the final completion status.
2	A decimal number indicating the number of terminals and mailboxes to which \$BRKTHRU successfully sent the message.
3	A decimal number indicating the number of terminals to which \$BRKTHRU failed to send the message because the write to the terminal(s) timed out.
4	A decimal number indicating the number of terminals to which \$BRKTHRU failed to send the message because the terminal(s) had set the NOBROADCAST characteristic (by using the DCL command SET TERMINAL/NOBROADCAST).

carcon

VMS Usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Carriage control specifier indicating the carriage control sequence that is to follow the message that \$BRKTHRU sends to the terminal(s). The **carcon** argument is a longword containing the carriage control specifier.

Refer to the *VAX/VMS I/O Reference Volume* for a list of the carriage control specifiers that may be specified in the **carcon** argument.

If the **carcon** argument is not specified, \$BRKTHRU uses a default value of 32, which represents a space in the ASCII character set. The message format resulting from this default value is a line feed, the message text, and a carriage return.

The **carcon** argument has no effect on message formatting specified by the **BRK\$M_SCREEN** flag in the **flags** argument. Refer to the description of this flag for more information.

System Service Descriptions

\$BRKTHRU

flags

VMS Usage: **mask_longword**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Flag bit mask specifying options for the \$BRKTHRU operation. The **flags** argument is a longword value that is the logical OR of each desired flag option.

Each flag option has a symbolic name; these names are defined by the \$BRKDEF macro. The following list describes these symbolic names:

Symbolic name	Description
BRK\$V_ERASE_LINES	<p>When specified with the BRK\$M_SCREEN flag, BRK\$V_ERASE_LINES causes a specified number of lines to be cleared from the screen before the message is displayed. When BRK\$M_SCREEN is not also specified, BRK\$V_ERASE_LINES is ignored.</p> <p>Unlike the other Boolean flags, BRK\$V_ERASE_LINES specifies a 1-byte integer in the range 0 to 24. It occupies the first byte in the longword flag mask. In coding the call to \$BRKTHRU, specify the desired integer value in the OR operation with any other desired flags.</p>
BRK\$M_SCREEN	<p>When specified, \$BRKTHRU sends screen-formatted messages as well as messages formatted through the use of the carcon argument. \$BRKTHRU sends screen-formatted messages to terminals with the DEC_CRT characteristic, and it sends messages formatted by carcon to those without the DEC_CRT characteristic. Terminals set the DEC_CRT characteristic by using the DCL command SET TERMINAL/DEC_CRT.</p> <p>A screen-formatted message is displayed at the top of the terminal screen, and the cursor is repositioned at the point it was prior to the broadcast message. However, the BRK\$V_ERASE_LINES and BRK\$M_BOTTOM flags also affect the display.</p>
BRK\$M_BOTTOM	<p>When BRK\$M_BOTTOM is specified and BRK\$M_SCREEN is also specified, \$BRKTHRU writes the message to the bottom of the terminal screen instead of the top. BRK\$M_BOTTOM is ignored if the BRK\$M_SCREEN flag is not set.</p>
BRK\$M_NOREFRESH	<p>When BRK\$M_NOREFRESH is specified, \$BRKTHRU, after writing the message to the screen, does not redisplay the last line of a read operation that was interrupted by the broadcast message. This flag is useful only when the BRK\$M_SCREEN flag is not specified, since BRK\$M_NOREFRESH is the default for screen-formatted messages.</p>

System Service Descriptions

\$BRKTHRU

Symbolic name	Description
BRK\$M_CLUSTER	Specifying BRK\$M_CLUSTER enables \$BRKTHRU to send the message to terminals or mailboxes on other VAX nodes in a VAXcluster. If BRK\$M_CLUSTER is not specified, \$BRKTHRU sends messages only to terminals or mailboxes on the local VAX node.

reqid

VMS Usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Class requestor identification, which identifies to \$BRKTHRU the application (or image) that is calling \$BRKTHRU. The **reqid** argument is this longword identification value.

The **reqid** argument is used by several VAX/VMS images that send messages to terminals and may be used by as many as 16 different user images as well.

When such an image calls \$BRKTHRU, specifying **reqid**, \$BRKTHRU notifies the terminal that this image desires to write to the terminal. This makes it possible for the terminal user to allow or prevent the image from writing to the terminal.

To prevent a particular image from writing to the user's terminal, the user simply mentions the image's name in the DCL command SET TERMINAL /NOBROADCAST=image-name. Note that (image-name) in this DCL command is the same as the value of the **reqid** argument that the image passed to \$BRKTHRU.

For example, a terminal user can prevent the VAX/VMS MAIL utility (which is an image) from writing to the terminal by issuing the DCL command SET TERMINAL/NOBROADCAST=MAIL.

The \$BRKDEF macro defines class names that are used by several VAX/VMS components. These components specify their class names by using the **reqid** argument in calls to \$BRKTHRU. The \$BRKDEF macro also defines 16 class names (BRK\$C_USER1 through BRK\$C_USER16) for the use of user images that call \$BRKTHRU. The following lists each class name and the component to which it corresponds:

Class Name	Component
BRK\$C_GENERAL	This class name is used by (1) the VAX/VMS image invoked by the DCL command REPLY and (2) the callers of the \$BRDCST service. This is the default if the reqid argument is not specified.
BRK\$C_PHONE	This class name is used by the VAX/VMS PHONE facility.
BRK\$C_MAIL	This class name is used by the VAX/VMS MAIL utility.

System Service Descriptions

\$BRKTHRU

Class Name	Component
BRK\$_DCL	This class name is used by the Digital Command Language (DCL) interpreter for the CTRL/T command, which displays the process status.
BRK\$_QUEUE	This class name is used by the VAX/VMS queue manager, which manages print and batch jobs.
BRK\$_SHUTDOWN	This class name is used by the VAX/VMS system shutdown image, which is invoked by the DCL command REPLY/ID=SHUTDOWN .
BRK\$_URGENT	This class name is used by the VAX/VMS image invoked by the DCL command REPLY/ID=URGENT .
BRK\$_USER1 through BRK\$_USER16	These class names can be used by user-written images.

timeout

VMS Usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Timeout value, which is the number of seconds that must elapse before an attempted write by \$BRKTHRU to a terminal is considered to have failed. The **timeout** argument is this longword value (in seconds).

Since \$BRKTHRU calls the \$QIO service to perform writes to the terminal, the timeout value specifies the number of seconds allotted to \$QIO to perform a single write to the terminal.

If the **timeout** argument is not specified, \$BRKTHRU uses a default value of 0 seconds, which specifies infinite time; in other words, \$BRKTHRU does not time out.

The value specified by **timeout** may be 0 or any number greater than 4; the numbers 1, 2, 3, and 4 are illegal.

When a terminal user has pressed **CTRL/S** or the **NO SCROLL** key, \$BRKTHRU cannot send a message to the terminal. In such a case, the value of **timeout** will probably be exceeded and the attempted write to the terminal will fail.

astadr

VMS Usage: **ast_procedure**
type: **procedure entry mask**
access: **call without stack unwinding**
mechanism: **by reference**

AST service routine to be executed after \$BRKTHRU has sent the message to the specified terminal(s). The **astadr** argument is the address of the entry mask of this routine.

If **astadr** is specified, the AST routine will execute at the same access mode as the caller of \$BRKTHRU.

System Service Descriptions

\$BRKTHRU

astprm

VMS Usage: **user_arg**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

AST parameter to be passed to the AST routine specified by the **astadr** argument. The **astprm** argument specifies this longword parameter.

DESCRIPTION

The calling process must have OPER privilege to send a message to either more than one terminal or to a terminal that is allocated to another user.

The calling process must have WORLD privilege to send a message to a specific user by specifying the **BRK\$_C_USERNAME** symbolic code for the **sndtyp** argument.

The \$BRKTHRU service permits the message text to be as long as 16,350 bytes; however, both the SYSGEN parameter **MAXBUF** and the caller's available process space may also affect the maximum length of the message text.

The \$BRKTHRU service operates by assigning a channel (by using the \$ASSIGN service) to the terminal and then writing to the terminal (by using the \$QIO service). When calling \$QIO, \$BRKTHRU specifies the **IO\$_WRITEVBLK** function code, together with **IO\$_M_BREAKTHRU**, **IO\$_M_CANCTRLO**, and (optionally) **IO\$_M_REFRESH** function modifiers.

The current state of the terminal determines if and when the broadcast message is displayed on the screen:

- If the terminal is reading when \$BRKTHRU sends the message, the read operation is suspended, the message is displayed, and then the line that was being read when the read operation was suspended is redisplayed (equivalent to the action produced by **CTRL/R**).
- If the terminal is writing when \$BRKTHRU sends the message, the message is displayed after the current write operation has completed.
- The message is not displayed if the terminal has set the **NOBROADCAST** characteristic for all images, or if the terminal has disabled the receiving of messages from the image that is issuing the \$BRKTHRU call (see the **reqid** argument).

After the message is displayed, the terminal is returned to the state it was in prior to receiving the message.

System Service Descriptions

\$BRKTHRU

CONDITION VALUES RETURNED

SS\$_NORMAL	Service successfully completed.
SS\$_ACCVIO	The message buffer, message buffer descriptor, device name string, or device name string descriptor cannot be read by the caller.
SS\$_BADPARAM	The message length exceeds 16,350 bytes; the process's buffered I/O byte count limit (BYTLM) quota is insufficient; or the message length exceeds the value specified by the SYSGEN parameter MAXBUF.
SS\$_EXQUOTA	The process has exceeded its buffer space quota and has disabled resource wait mode with the Set Resource Wait Mode (\$SETRWM) service.
SS\$_INSFMEM	Insufficient system dynamic memory is available to complete the request and the process has disabled resource wait mode with the Set Resource Wait Mode (\$SETRWM) service.
SS\$_NONLOCAL	Warning. The device is on a remote node.
SS\$_NOOPER	The process does not have the necessary OPER privilege.
SS\$_NOSUCHDEV	Warning. The specified terminal does not exist, or it cannot receive the message.

CONDITION VALUES RETURNED IN THE I/O STATUS BLOCK

Any condition values returned by the \$ASSIGN, \$FAO, \$GETDVI, \$GETJPL, or \$QIO services.

\$BRKTHRUW—Breakthrough and Wait

The Breakthrough and Wait service sends a message to one or more terminals.

The \$BRKTHRUW service operates synchronously; that is, it returns to the caller after the message has been sent to the specified terminal(s).

For asynchronous operations, use the Breakthrough (\$BRKTHRU) service; \$BRKTHRU returns to the caller after queuing the message request, without waiting for the message to be delivered.

In all other respects, \$BRKTHRUW is identical to \$BRKTHRU. Refer to the documentation of \$BRKTHRU for all other information about the \$BRKTHRUW service.

For additional information about system service completion, refer to the documentation of the Synchronize (\$SYNCH) service and to Section 2.5 in Part I.

The \$BRKTHRU and \$BRKTHRUW services supersede the Broadcast (\$BRDCST) service. New code should be written using \$BRKTHRU or \$BRKTHRUW, and old code using \$BRDCST should be converted to use \$BRKTHRU or \$BRKTHRUW as convenient. \$BRDCST is now an obsolete system service and will no longer be enhanced.

FORMAT

SYS\$BRKTHRUW *[efn],msgbuf[,sendto][,sndtyp]
[,iosb][,carcon][,flags][,reqid]
[,timeout][,astadr][,astprm]*

System Service Descriptions

\$CANCEL

\$CANCEL—Cancel I/O on Channel

The Cancel I/O On Channel service cancels all pending I/O requests on a specified channel. In general, this includes all I/O requests that are queued as well as the request currently in progress.

FORMAT **SYSCANCEL** *chan*

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENT *chan*

VMS Usage: **channel**
type: **word (unsigned)**
access: **read only**
mechanism: **by value**

I/O channel on which I/O is to be canceled. The **chan** argument is a longword containing the channel number.

DESCRIPTION

To cancel I/O on a channel, the access mode of the calling process must be equal to or more privileged than the access mode of the process that made the original channel assignment.

The \$CANCEL service requires system dynamic memory and uses the process's buffered I/O limit (BIOLM) quota.

When a request currently in progress is canceled, the driver is notified immediately. Actual cancellation may or may not occur immediately, depending on the logical state of the driver. When cancellation does occur, the action taken for I/O in progress is similar to that taken for queued requests, that is:

- 1 The specified event flag is set.
- 2 The first word of the I/O status block, if specified, is set to SS\$_CANCEL if the I/O request is queued or to SS\$_ABORT if the I/O is in progress.
- 3 The AST, if specified, is queued.

Proper synchronization between this service and the actual canceling of I/O requests requires the issuing process to wait for I/O completion in the normal manner and then note that the I/O has been canceled.

If the I/O operation is a virtual I/O operation involving a disk or tape ACP, the I/O cannot be canceled. In the case of a magnetic tape, however, cancellation may occur if the device driver is hung.

System Service Descriptions

\$CANCEL

Outstanding I/O requests are automatically canceled at image exit.

CONDITION VALUES RETURNED

SS\$_NORMAL	Service successfully completed.
SS\$_EXQUOTA	The process has exceeded its buffered I/O limit (BIOLM) quota.
SS\$_INSFMEM	Insufficient system dynamic memory is available to cancel the I/O.
SS\$_JVCHAN	An invalid channel was specified, that is, a channel number of 0 or a number larger than the number of channels available.
SS\$_NOPRIV	The specified channel is not assigned or was assigned from a more privileged access mode.

System Service Descriptions

\$SCANEXH

\$SCANEXH—Cancel Exit Handler

The Cancel Exit Handler service deletes an exit control block from the list of control blocks for the calling access mode. Exit control blocks are declared by the Declare Exit Handler (\$DCLEXH) service and are queued according to access mode in a last-in first-out order.

FORMAT	SY\$SCANEXH [<i>desblk</i>]
---------------	--------------------------------------

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENT

desblk

VMS Usage: **exit_handler_block**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Control block describing the exit handler to be canceled. The ***desblk*** is the address of this control block. If ***desblk*** is not specified or is specified as 0, all exit control blocks are canceled for the current access mode.

CONDITION VALUES RETURNED

SS\$_NORMAL

Service successfully completed.

SS\$_ACCVIO

The first longword of the exit control block or the first longword of a previous exit control block in the list cannot be read by the caller, or the first longword of the preceding control block cannot be written by the caller.

SS\$_IVSSRQ

The call to the service is invalid because it was made from kernel mode.

SS\$_NOHANDLER

The specified exit handler does not exist.

\$CANTIM—Cancel Timer

The Cancel Timer Request service cancels all or a selected subset of the Set Timer requests previously issued by the current image executing in a process. Cancellation is based on the request identification specified in the Set Timer (\$SETIMR) service. If more than one timer request was given the same request identification, all requests with that request identification are canceled.

FORMAT **SYSCANTIM** [*reqidt*],[*acmode*]

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS *reqidt*

VMS Usage: **user_arg**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Request identification of the timer request(s) to be canceled. The **reqidt** argument is a longword containing this identification. If **reqidt** is specified as 0 (the default), all timer requests are canceled.

acmode

VMS Usage: **access_mode**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Access mode of the request(s) to be canceled. The **acmode** argument is a longword containing the access mode. The \$PSLDEF macro defines the following symbols for the four access modes:

Symbol	Access mode
PSL\$C_KERNEL	Kernel mode
PSL\$C_EXEC	Executive mode
PSL\$C_SUPER	Supervisor mode
PSL\$C_USER	User mode

The most privileged access mode used is the access mode of the caller. Only those timer requests issued from an access mode equal to or less privileged than the resultant access mode are canceled.

System Service Descriptions

SCANTIM

DESCRIPTION The calling process can only cancel timer requests that were issued by a process whose access mode is equal to or less privileged than that of the calling process.

Canceled timer requests are restored to the process's quota for timer queue entries (TQELM quota).

Outstanding timer requests are automatically canceled at image exit.

**CONDITION
VALUES
RETURNED**

SS\$_NORMAL

Service successfully completed.

\$SCANWAK—Cancel Wakeup

The Cancel Wakeup service removes all scheduled wake-up requests for a process from the timer queue, including those made by the caller or by other processes. Scheduled wake-up requests are made with the Schedule Wakeup (\$SCHDWK) service.

FORMAT **SY\$SCANWAK** [*pidadr*],[*prcnam*]

RETURNS

VMS Usage: **cond_value**
 type: **longword (unsigned)**
 access: **write only**
 mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS

pidadr

VMS Usage: **process_id**
 type: **longword (unsigned)**
 access: **modify**
 mechanism: **by reference**

Process identification (PID) of the process for which wakeups are to be canceled. The *pidadr* is the address of a longword specifying the PID.

prcnam

VMS Usage: **process_name**
 type: **character-coded text string**
 access: **read only**
 mechanism: **by descriptor—fixed length string descriptor**

Name of the process for which wakeups are to be canceled. The *prcnam* argument is the address of a character string descriptor pointing to the process name string.

VAX/VMS interprets the UIC group number of the calling process as part of the process name; the names of processes are unique to UIC groups. Because of this, the *prcnam* argument can only be used on behalf of processes in the same group as the calling process.

DESCRIPTION

Depending on the operation, use of \$SCANWAK may require the calling process to have certain privilege:

- GROUP privilege is required to cancel wakeup requests issued for other processes in the same group unless the process has the same UIC
- WORLD privilege is required to cancel wakeup requests issued for any process in the system

Canceled wake-up requests are restored to the process's AST limit (ASTLM) quota.

System Service Descriptions

\$SCANWAK

If neither the **pidadr** nor **prcnam** arguments are specified, scheduled wake-up requests for the calling process are canceled.

Pending wake-up requests issued by the current image are automatically canceled at image exit.

This service only cancels wake-up requests that have been scheduled; it does not cancel wake-up requests made with the Wake Process from Hibernation (\$WAKE) service.

CONDITION VALUES RETURNED

SS\$_NORMAL

Service successfully completed.

SS\$_ACCVIO

The process name string or string descriptor cannot be read by the caller, or the process identification cannot be written by the caller.

SS\$_IVLOGNAM

The process name string has a length of 0 or has more than 15 characters.

SS\$_NONEXPR

Warning. The specified process does not exist, or an invalid process identification was specified.

SS\$_NOPRIV

The process does not have the privilege to cancel wakeups for the specified process.

\$CHANGE_ACL—Change Access Control List

The Change Access Control List service creates or modifies an object's access control list.

FORMAT **SY\$CHANGE_ACL** *[chan]* , *objtyp* , *[objnam]* , *itmlst* , *[acmode]* , *[nullarg]* , *[contxt]*

RETURNS VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS *chan*

VMS Usage: **channel**
type: **word (unsigned)**
access: **read only**
mechanism: **by value**

Number of the I/O channel assigned to the object whose ACL is modified when \$CHANGE_ACL completes execution. The **chan** argument is a word containing the number of the channel. If **objnam** is specified, **chan** must be omitted or specified as zero.

objtyp

VMS Usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Type of object whose ACL is modified when \$CHANGE_ACL completes execution. The **objtyp** argument is the address of a longword containing a value indicating whether the object is a file or a device. The symbols are defined in the system macro library (\$ACLDEF).

ACL\$_DEVICE	Object is a device
ACL\$_FILE	Object is a Files-11 structure level 2 file
ACL\$_GROUP_GLOBAL_SECTION	Object is a group global section
ACL\$_LOGICAL_NAME_TABLE	Object is a logical name table
ACL\$_SYSTEM_GLOBAL_SECTION	Object is a system global section

objnam

VMS Usage: **char_string**
type: **character-coded text string**

System Service Descriptions

\$CHANGE_ACL

access: **read only**

mechanism: **by descriptor—fixed length string descriptor**

Name of the object whose ACL is modified when \$CHANGE_ACL completes execution. The **objnam** argument is the address of a descriptor pointing to a character text string containing the name of the object. The maximum length of **objnam** depends on the object.

itmlst

VMS Usage: **item_list_3**

type: **longword (unsigned)**

access: **read only**

mechanism: **by reference**

Modifications to be made to the ACL when \$CHANGE_ACL completes execution. The **itmlst** argument is the address of a variable length data structure defining the changes to be made. The data structure consists of three elements for each item code, as follows:

code	buflen
bufadr	
unused	

ZK-1701-84

End the item list with a longword containing the value zero.

buflen Word containing the number of bytes in the buffer pointed to by **bufadr**

code Word containing the item code

bufadr Longword containing the address of a buffer for information being passed to or from \$CHANGE_ACL

The third longword of the standard item list entry (the return length address) is not used by \$CHANGE_ACL and should be zero.

The symbols for the item codes are defined in the system macro library (\$ACLDEF) as follows:

ACL\$_ACLENGTH Returns the size, in bytes, of the object's ACL. **Bufadr** points to a longword that contains the size.

ACL\$_ADDACLENT Adds an ACE to the beginning of the ACL when **ctxt** is 0, to the end of the ACL when **ctxt** is -1, or at a location pointed to by a prior **ACL\$_FNDACETYP** or **ACL\$_FNDACLENT**. **Bufadr** points to a variable length data structure containing the ACE to be added. You can add more than one ACE with **ACL\$_ADDACLENT**; however, **buflen** must contain the total size of all ACEs contained in the buffer.

ACL\$_DELACLENT Deletes the ACE pointed to by **bufadr** or if **bufadr** is specified as zero, the ACE pointed to by a prior **ACL\$_FNDACETYP** or **ACL\$_FNDACLENT**.

System Service Descriptions

\$CHANGE_ACL

ACL\$C_DELETEACL	Deletes the entire ACL with the exception of protected ACEs.
ACL\$C_FNDACETYP	Locates an ACE of the type pointed to by bufadr .
ACL\$C_FNDACLENT	Locates the ACE pointed to by bufadr .
ACL\$C_RLOCK_ACL	Obtains a read lock on an object in order to lock out all writers to the object's ACL. Regardless of the caller's mode, ACL locks are user mode locks so that all access modes will interlock ACLs correctly.
ACL\$C_WLOCK_ACL	Obtains an exclusive lock on an object in order to lock out all other readers and writers to the object's ACL. Regardless of the caller's mode, ACL locks are user mode locks so that all access modes will interlock ACLs correctly.
ACL\$C_MODACLENT	Replaces the ACE pointed to by a prior ACL\$C_FNDACETYP or ACL\$C_FNDACLENT with the ACE pointed to by bufadr .
ACL\$C_READACE	Reads the ACE pointed to by ACL\$C_FNDACETYP or ACL\$C_FNDACLENT into the buffer pointed to by bufadr .
ACL\$C_READACL	Reads the object's ACL. Contxt should be initially set to zero. Complete ACEs are read into the buffer pointed to by bufadr .
ACL\$C_UNLOCK_ACL	Releases the lock obtained with ACL\$C_RLOCK_ACL or ACL\$C_WLOCK_ACL.

When you add an ACE with ACL\$C_ADDACLENT or locate an ACE with ACL\$C_FNDACETYP or ACL\$C_FNDACLENT, \$CHANGE_ACL searches the ACL for a match for the ACE in the ACE buffer. \$CHANGE_ACL does not always make a match based on the entire ACE buffer; instead, the ACE type determines how \$CHANGE_ACL makes a match.

- A default protection ACE (ACE\$C_DIRDEF) matches only on the type field (ACE\$B_TYPE). An ACL can have only one default protection ACE because \$CHANGE_ACL stops searching once it locates a match.
- An identifier ACE (ACE\$C_KEYID) matches on the flags (ACE\$W_FLAGS) and identifier (ACE\$L_KEY) fields.
- An alarm ACE (ACE\$C_ALARM) matches on the flags (ACE\$W_FLAGS) and access mask (ACE\$L_ACCESS) fields.
- All other ACE types match on the entire ACE buffer.

Because \$CHANGE_ACL uses these matching rules, adding an ACE sometimes results in the replacement of another ACE. For example, if you add an identifier ACE with the same flags and identifier fields but a different access mask, the new ACE replaces the old ACE. When you add an ACE on the top of an ACL, \$CHANGE_ACL deletes any matching ACE since it will not be seen. If you add an ACE below a matching ACE in an ACL, the added ACE has no effect since it will not be seen.

acmode

VMS Usage: **access_mode**
type: **longword (unsigned)**
access: **read only**

System Service Descriptions

\$CHANGE_ACL

mechanism: **by reference**

Access mode to use in checking file access protection. The **acmode** argument is the address of a longword containing the access mode. **Acmode** defaults to kernel mode; however, the system compares **acmode** against the caller's access mode and uses the least privileged mode.

The access modes and their symbolic names are defined in the system macro library (\$PSLDEF) as follows:

Symbol	Access Mode
PSL\$C_USER	User
PSL\$C_SUPER	Supervisor
PSL\$C_EXEC	Executive
PSL\$C_KERNEL	Kernel

nullarg

VMS Usage: **null_arg**

type: **longword (unsigned)**

access: **read only**

mechanism: **by value**

Place-holding argument. This argument is reserved to DIGITAL.

ctxt

VMS Usage: **context**

type: **longword (unsigned)**

access: **modify**

mechanism: **by reference**

Context value that points to an ACE. The **ctxt** argument is the address of a longword containing the context value.

DESCRIPTION	The Change Access Control List service creates or modifies an object's ACL. See \$FORMAT_ACL for information on the various types of ACLs and their associated formats. See \$PARSE_ACL for information on how to convert an ASCII string to an ACE.
--------------------	--

CONDITION VALUES RETURNED

SS\$_NORMAL	Service successfully completed.
SS\$_ACCVIO	The string or its descriptor cannot be read by the caller, or the buffer descriptor cannot be read by the caller, or the buffer cannot be written by the caller, or the buffer is too small to hold the ACL entry.
SS\$_BADPARAM	Invalid object type, attribute code, item size, or access mode is specified.
SS\$_INSFARG	Objtyp is not specified, or neither chan nor objnam are specified.
SS\$_IVACL	The format of the access control list entry is not valid.

\$CHECK_ACCESS—Check Access

The Check Access system service determines, on behalf of a third-party user, whether the user can access the object specified.

FORMAT **SY\$CHECK_ACCESS** *objtyp,objnam,usnam*
,itmlst

RETURNS

VMS Usage: **cond_value**
 type: **longword (unsigned)**
 access: **write only**
 mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS

objtyp

VMS Usage: **longword_unsigned**
 type: **longword (unsigned)**
 access: **read only**
 mechanism: **by reference**

Type of object being accessed. The *objtyp* argument is the address of a longword containing a value specifying the type of object. The symbols are defined in the system macro library (\$ACLDDEF).

ACL\$C_DEVICE	Object is a device
ACL\$C_FILE	Object is a Files-11 structure level 2 file
ACL\$C_GROUP_GLOBAL_SECTION	Object is a group global section
ACL\$C_LOGICAL_NAME_TABLE	Object is a logical name table
ACL\$C_SYSTEM_GLOBAL_SECTION	Object is a system global section

objnam

VMS Usage: **char_string**
 type: **character-coded text string**
 access: **read only**
 mechanism: **by descriptor—fixed length string descriptor**

Name of the object being accessed. The *objnam* argument is the address of a descriptor pointing to a character text string containing the name of the object. The maximum length of *objnam* depends on the object.

usnam

VMS Usage: **char_string**
 type: **character-coded text string**
 access: **read only**

System Service Descriptions

\$CHECK_ACCESS

mechanism: **by descriptor—fixed length string descriptor**

Name of the user attempting access. The **usnam** argument is the address of a descriptor pointing to a character text string containing the user name of the user attempting to gain access to the specified object. The user name string may contain a maximum of 12 alphanumeric characters.

itmlst

VMS Usage: **item_list_3**

type: **longword (unsigned)**

access: **read only**

mechanism: **by reference**

Attributes describing how the object is to be accessed and information returned after \$CHECK_ACCESS performs the protection check (for instance, security alarm information).

For each item code, include a set of four elements as shown below and end the list with a longword containing the value 0 (CHP\$_END).

code	buflen
bufadr	
retlenadr	

ZK-1703-84

buflen	Word containing the number of bytes in the buffer pointed to by bufadr .
code	Word containing the item code. The item codes are defined in the system macro library (\$CHPDEF).
bufadr	Longword containing the address of a buffer used to pass information to or receive information from SYS\$CHECK_ACCESS.
retlenadr	Longword containing the address of a word-long buffer in which SYS\$CHECK_ACCESS writes the number of bytes written to the buffer pointed to by bufadr . If the buffer pointed to by bufadr is used to pass information to SYS\$CHECK_ACCESS, retlenadr is ignored but must be included.

All items are optional. If the access type item code (CHP\$_ACCESS) is not specified, read access is assumed.

The following sections describe the item codes used with \$CHECK_ACCESS. The first list of item codes defines the type of access desired. The second list of item codes allows you to determine which rights and privileges were used to access the object. The item codes are defined in the system macro library (\$CHPDEF).

System Service Descriptions

\$CHECK_ACCESS

Input Items—Type of Access Desired

Item Identifier	Data Type	Description
CHP\$_ACCESS	longword	Bit mask representing the type of access desired (\$ARMDEF)
CHP\$_ACMODE	byte	Accessor's processor access mode (\$PSLDEF)
CHP\$_FLAGS	longword	Accessor's access to the object

CHP\$_ACCESS

Any bits not specified by CHP\$_ACCESS are assumed to be clear, which grants access. (The default definitions can be found in the \$ARMDEF macro.)

CHP\$_ACMODE

The access modes and their symbolic names are defined in the system macro library (\$PSLDEF) as follows:

Symbol	Access Mode
PSL\$_USER	User
PSL\$_SUPER	Supervisor
PSL\$_EXEC	Executive
PSL\$_KERNEL	Kernel

CHP\$_FLAGS

The following symbol values are offsets to the bits within the longword. You can also obtain the values as masks with the appropriate bit set by using the prefix CHP\$_M rather than CHP\$_V. The symbols are defined only in the system macro library (\$CHPDEF).

Symbol	Access
CHP\$_V_READ	Accessor has read access.
CHP\$_V_WRITE	Accessor has write access.
CHP\$_V_USEREADALL	Accessor is eligible for READALL privilege.

Output Items—Information Returned

Item Identifier	Data Type	Description
CHP\$_AUDITNAME	string	Character string containing the audit record
CHP\$_ALARMNAME	string	Character string containing the alarm record
CHP\$_MATCHEDACE	block	The ACE in object's ACL that allowed or denied the accessor access to the object
CHP\$_PRIVUSED	longword	Mask of flags representing privileges used to gain the requested access

System Service Descriptions

\$CHECK_ACCESS

CHP\$_ALARMNAME

If the object does not have security alarms enabled, SYS\$CHECK_ACCESS returns **retlenadr** as 0.

CHP\$_AUDITNAME

If the object does not have auditing enabled, SYS\$CHECK_ACCESS returns **retlenadr** as 0.

CHP\$_MATCHEDACE

Variable-length data structure containing the ACE in the object's ACL that allowed or denied the accessor access to the object. The SYS\$FORMAT_ACL system service lists each ACE type and its format.

CHP\$_PRIVUSED

The symbol values used as offsets to the bits within the longword are

Symbol	Meaning
CHP\$_SYSPRV	SYSPRV was used to gain the requested access
CHP\$_GRPPRV	GRPPRV was used to gain the requested access
CHP\$_BYPASS	BYPASS was used to gain the requested access
CHP\$_READALL	READALL was used to gain the requested access

You can also obtain the values as masks with the appropriate bit set by using the prefix **CHP\$M** rather than **CHP\$V**. The symbols are defined in the system macro library (**\$CHPDEF**).

DESCRIPTION

The Check Access system service is used to check access to an object on behalf of a third-party process. One use of the \$CHECK_ACCESS service might be for a file server which uses the service to check the protection attributes of a process (the third-party accessor) attempting access to a file (the object).

If the accessor can access the object, SYS\$CHECK_ACCESS returns the **SS\$_NORMAL** status code; otherwise, SYS\$CHECK_ACCESS returns **SS\$_NOPRIV**.

The arguments accepted by this service specify the name and type of object being accessed, the name of the user requesting access to the object, the type of access desired, and the type of information returned.

Alarm name strings are returned if an alarm record is to be written. A nonzero string length (as returned in the item descriptor) specifies the presence of an alarm request; if none is requested, a zero length is returned. Note that alarms may be requested whether the protection check succeeds or fails.

System Service Descriptions

\$CHECK_ACCESS

CONDITION VALUES RETURNED

SS\$_NORMAL

Service successfully completed; the desired access is granted.

SS\$_NOPRIV

Service completed; the desired access is not granted.

SS\$_ACCVIO

The item list cannot be read by the caller, or one of the buffers specified in the item list cannot be written by the caller.

\$CHKPRO—Check Access Protection

The Check Access Protection system service determines whether an accessor with the specified rights and privileges can access an object with the specified attributes.

FORMAT **SY\$CHKPRO** *itmlst*

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS *itmlst*

VMS Usage: **item_list_3**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Protection attributes of the object and the rights and privileges of the accessor used when \$CHKPRO determines if the accessor can access the object. The *itmlst* argument is the address of an item list of descriptors used to specify the protection attributes of the object and the rights and privileges of the accessor.

For each item code, include a set of four elements as shown below and end the list with a longword containing the value of zero (CHP\$_END).

code	buflen
bufadr	
retlenadr	

ZK-1703-84

buflen	Word containing the number of bytes in the buffer pointed to by bufadr .
code	Word containing the item code. The item codes defined in the system macro library (\$ACLDEF).
bufadr	Longword containing the address of a buffer used to pass information to or receive information from SY\$CHKPRO.

System Service Descriptions

\$CHKPRO

retlenadr Longword containing the address of a word-long buffer in which SYSSCHKPRO writes the number of bytes written to the buffer pointed to by **bufadr**. If the buffer pointed to by **bufadr** is used to pass information to SYSSCHKPRO, **retlenadr** is ignored but must be included.

All items are optional. Specifying any particular protection attribute causes that protection check to be made; any protection attribute not specified is not checked. Rights and privileges specified are used as needed. If a protection check requires any right or privilege not specified in the item list, the right or privilege of the caller's process is used.

The following sections describe the item codes used with \$CHKPRO. The first list of item codes define the accessor's rights and privileges. The second list of item codes define the object's protection attributes. The third list of item codes allow you to determine which rights and privileges were used to access the object. The item codes are defined in the system macro library (\$CHPDEF).

Input Items—Accessor's Rights and Privileges

Item Identifier	Data Type	Description
CHP\$_ACCESS	longword	Bit mask representing the type of access desired. (\$ARMDEF)
CHP\$_ACMODE	byte	Accessor's processor access mode.
CHP\$_ADDRIGHTS	vector	Additional rights list segment to be appended to existing rights list.
CHP\$_FLAGS	longword	Accessor's access to the object.
CHP\$_PRIV	quadword	Accessor's privilege mask.
CHP\$_RIGHTS	vector	Accessor's rights list.

System Service Descriptions

\$CHKPRO

CHP\$_ACCESS

Be aware that the \$CHKPRO service does not interpret the bits in the access mask; instead, it compares them against the object's protection mask (CHP\$_PROT). Any bits not specified by CHP\$_ACCESS or CHP\$_PROT are assumed to be clear, which grants access.

CHP\$_ACMODE

The access modes and their symbolic names are defined in the system macro library (\$PSLDEF) as follows:

Symbol	Access Mode
PSL\$_USER	User
PSL\$_SUPER	Supervisor
PSL\$_EXEC	Executive
PSL\$_KERNEL	Kernel

CHP\$_ADDRIGHTS

Each entry of the rights list is a quadword data structure consisting of a longword containing the identifier value, followed by a longword containing a mask identifying the attributes of the holder. SYS\$CHKPRO ignores the attributes.

A maximum of 11 rights descriptors is allowed. If you specify CHP\$_ADDRIGHTS without specifying CHP\$_RIGHTS, the accessor's rights list consists of the rights list specified by the CHP\$_ADDRIGHTS item code(s) and the rights list of the current process.

If you specify CHP\$_RIGHTS and CHP\$_ADDRIGHTS:

- 1 CHP\$_RIGHTS must come first.
- 2 The accessor's UIC is the identifier of the first entry in the rights list specified by the CHP\$_RIGHTS item code.
- 3 The accessor's rights list consists of the rights list specified by the CHP\$_RIGHTS item code and the CHP\$_ADDRIGHTS item code(s).

CHP\$_FLAGS

The following symbol values are offsets to the bits within the longword. You can also obtain the values as masks with the appropriate bit set by using the prefix CHP\$_M rather than CHP\$_V. The symbols are defined only in the system macro library (\$CHPDEF).

Symbol	Access
CHP\$_V_READ	Accessor is making a read access
CHP\$_V_WRITE	Accessor is making a write access
CHP\$_V_USEREADALL	Accessor is eligible for READALL privilege

Since the access mask (CHP\$_ACCESS) is not interpreted by \$CHKPRO, CHP\$_FLAGS is used to determine whether the accessor is making a read and/or write access to the object.

System Service Descriptions

\$CHKPRO

CHP\$_PRIV

To form the symbolic names for the bits in the privilege mask, preface the name of the privileges with PRV\$_. For example, the bit associated with the BYPASS privilege is PRV\$_BYPASS. The privilege symbols are defined in the system macro library (\$PRVDEF).

CHP\$_RIGHTS

The accessor's UIC is the identifier of the first entry in the rights list. The accessor's rights list consists of the rights list specified by CHP\$_RIGHTS and optionally the rights list specified by CHP\$_ADDRIGHTS item code(s).

Input Items—Object's Protection Attributes

Item Identifier	Data Type	Description
CHP\$_ACL	vector	Object's access control list
CHP\$_MODE	byte	Object's owner access mode
CHP\$_MODES	quadword	Object's access mode protection
CHP\$_OWNER	longword	Object's owner UIC
CHP\$_PROT	vector	Object's "SOGW" protection mask

CHP\$_ACL

The buffer address, bufadr, specifies a buffer containing one or more ACE. The number that specifies the length of the CHP\$_ACL buffer, **buflen**, must be equal to the sum of all ACE lengths. The format of the ACE structure depends on the value of the second byte in the structure, which specifies the ACE type. The SYS\$FORMAT_ACL system service description describes each ACE type and its format.

The CHP\$_ACL item may be specified multiple times to point to multiple segments of an access control list. A maximum of 20 segments may be specified. The segments are processed in the order specified.

CHP\$_MODE

The access modes of the object's owner and their symbolic names are defined in the system macro library (\$PSLDEF) as follows:

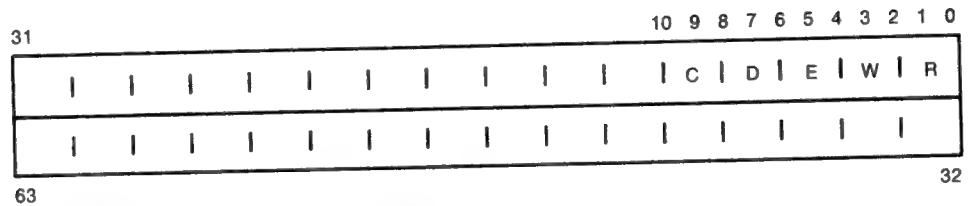
Symbol	Access Mode
PSL\$_USER	User
PSL\$_SUPER	Supervisor
PSL\$_EXEC	Executive
PSL\$_KERNEL	Kernel

CHP\$_MODES

Specify a two-bit access mode as shown in CHP\$_MODE for each possible access type. The following figure illustrates the format of an access mode vector.

System Service Descriptions

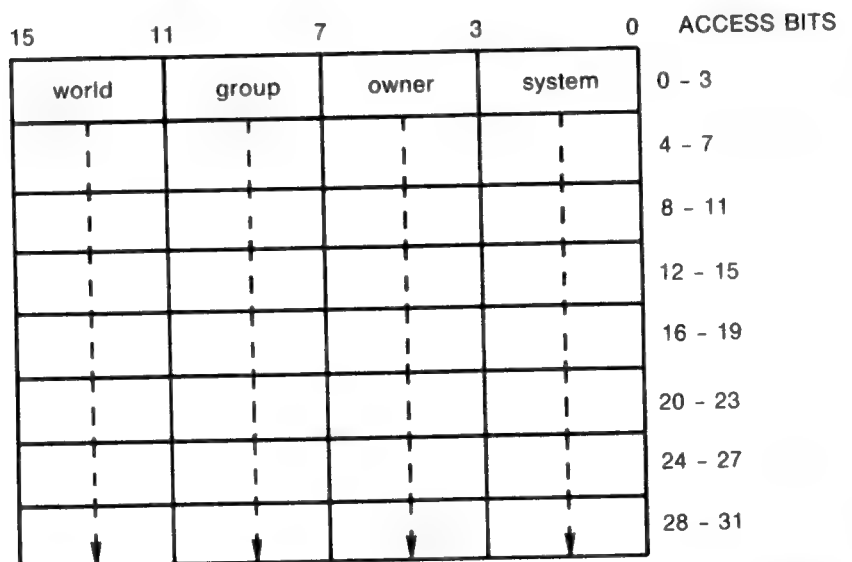
\$CHKPRO



Each pair of the bits in the access mode vector represent the access mode for the particular type of access. For example, bits <6:7> represent the access mode value used to check for delete access.

CHP\$_PROT

The following figure depicts the format for describing the object's protection.



The first word contains the first four protection bits for each field, the second word the next four protection bits, and so on. If a bit is clear, access is granted. By convention, the first five protection bits are (from right to left in each field of the first word) read, write, execute, delete, and (in the low-order bit in each field of the second word) control access. CHP\$_PROT may be specified in increments of words; if a short buffer is given, zeroes are assumed for the remainder.

\$CHKPRO compares the low-order four bits of CHP\$_ACCESS against one of the four bit fields in the low-order word of CHP\$_PROT, the next four bits of CHP\$_ACCESS against one of the four bit fields in the next word of CHP\$_PROT, and so on. \$CHKPRO chooses a field of CHP\$_PROT based on the privileges specified for the accessor (CHP\$_PRIV), and the UICs of the accessor (CHP\$_RIGHTS and/or CHP\$_ADDRIGHTS) and the object's owner (CHP\$_OWNER).

System Service Descriptions

\$CHKPRO

Output Items—Information Returned

Item Identifier	Data Type	Description
CHP\$_ALARMNAME	string	Character string containing the alarm record
CHP\$_MATCHEDACE	block	Contains the ACE in object's ACL that allowed the accessor to access the object
CHP\$_PRIVUSED	longword	Mask of flags representing privileges used to gain the requested access

CHP\$_ALARMNAME

If the object does not have security alarms enabled, SYS\$CHKPRO returns **retlenadr** as zero.

CHP\$_PRIVUSED

The symbol values used as offsets to the bits within the longword are:

Symbol	Meaning
CHP\$_SYSPRV	Used SYSPRV to gain the requested access
CHP\$_GRPPRV	Used GRPPRV to gain the requested access
CHP\$_BYPASS	Used BYPASS to gain the requested access
CHP\$_READALL	Used READALL to gain the requested access

You can also obtain the values as masks with the appropriate bit set by using the prefix **CHP\$M** rather than **CHP\$V**. The symbols are defined in the system macro library (**\$CHPDEF**).

DESCRIPTION

The Check Access Protection service invokes the system's access protection check, allowing layered products and other subsystems to build protected structures within themselves whose protection is consistent with the protection facilities provided by the base system. It also allows a privileged subsystem to perform protection checks on behalf of some requestor.

If the accessor can access the object, SYS\$CHKPRO returns the **SS\$_NORMAL** status code; otherwise, SYS\$CHKPRO returns **SS\$_NOPRIV**.

The arguments accepted by this service specify the protection of the object being accessed, the rights and privileges of the accessor, and the type of access desired. Because of the diverse and extensible nature of protections available in VAX/VMS, the arguments are presented in an item list.

When a protection check is to be invoked on the behalf of another process, the privilege mask (**CHP\$_PRIV**) is almost always mandatory, since it is used for all types of protection checks.

Alarm name strings are returned if an alarm record is to be written. A non-zero string length (as returned in the item descriptor) specifies the presence of an alarm request; if none is requested, a zero length is returned. Note that alarms may be requested whether the protection check succeeds or fails.

System Service Descriptions

\$CHKPRO

CONDITION VALUES RETURNED

SS\$_NORMAL

Service successfully completed; the desired access is granted.

SS\$_NOPRIV

Service completed; the desired access is not granted.

SS\$_ACCVIO

The item list cannot be read by the caller, or one of the buffers specified in the item list cannot be written by the caller.

SS\$_BADPARAM

Invalid argument.

SS\$_JVACL

An invalid ACL segment was supplied with the CHP\$_ACL item.

System Service Descriptions

\$CLREF

\$CLREF—Clear Event Flag

The Clear Event Flag service clears (sets to 0) an event flag in a local or common event flag cluster.

FORMAT **SY\$CLREF** *efn*

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENT

efn

VMS Usage: **ef_number**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Number of the event flag to be cleared. The *efn* argument is a longword containing this number.

CONDITION VALUES RETURNED

SS\$_WASCLR

Service successfully completed. The specified event flag was previously 0.

SS\$_WASSET

Service successfully completed. The specified event flag was previously 1.

SS\$_ILLEFC

An illegal event flag number was specified.

SS\$_UNASEFC

The process is not associated with the cluster containing the specified event flag.

\$CMEXEC—Change to Executive Mode

The Change to Executive Mode service changes the access mode of the calling process to executive mode. This service allows a process to change its access mode to executive, execute a specified routine, and then return to the access mode in effect before the call was issued.

FORMAT **SY\$CMEXEC** *routin* [,*arglst*]

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS *routin*

VMS Usage: **procedure**
type: **procedure entry mask**
access: **call without stack unwinding**
mechanism: **by reference**

Routine to be executed while the process is in executive mode. The *routin* argument is the address of the entry point to this routine.

arglst

VMS Usage: **arg_list**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Argument list to be passed to the routine specified by the *routin* argument. The *arglst* argument is the address of this argument list.

DESCRIPTION

To call this service, the process must either have CMEXEC or CMKRNL privilege or be currently executing in executive or kernel mode.

The \$CMEXEC service uses standard procedure calling conventions to pass control to the specified routine. If no argument list is specified, the argument pointer (AP) contains a 0, unless it is modified by the caller. However, to conform to the VAX procedure calling standard, you must not omit the *arglst* argument.

When the \$CMEXEC service is used, the system service dispatcher modifies both R0 and R1 before entry into the target routine. The specified routine must exit with a RET instruction and should place a status value in R0 before returning.

System Service Descriptions

\$CMEXEC

All of the Change Mode system services are intended to allow for the execution of a routine at an access mode more (not less) privileged than the access mode from which the call is made. If \$CMEXEC is called while a process is executing in kernel mode, the routine specified by the **routine** argument executes in kernel mode, not executive mode.

CONDITION VALUES RETURNED

SS\$_NOPRIV

The process does not have the privilege to change mode to executive.

All other values

All other values are returned by the routine executed.

\$CMKRNL—Change to Kernel Mode

The Change to Kernel Mode service changes the access mode of the calling process to kernel mode. This service allows a process to change its access mode to kernel, execute a specified routine, and then return to the access mode in effect before the call was issued.

FORMAT **SY\$CMKRNL** *routine* [,*arglst*]

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS *routine*

VMS Usage: **procedure**
type: **procedure entry mask**
access: **call without stack unwinding**
mechanism: **by reference**

Routine to be executed while the process is in kernel mode. The **routine** argument is the address of the entry point to this routine.

arglst

VMS Usage: **arg_list**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Argument list to be passed to the routine specified by the **routine** argument. The **arglst** argument is the address of this argument list.

DESCRIPTION

To call the \$CMKRNL service, a process must either have CMKRNL privilege or be currently executing in executive or kernel mode.

The \$CMKRNL service uses standard procedure calling conventions to pass control to the specified routine. If no argument list is specified, the argument pointer (AP) contains a 0, unless it is modified by the caller. However, to conform to the VAX procedure calling standard, you must not omit the **arglst** argument.

When the \$CMKRNL service is used, the system service dispatcher modifies both R0 and R1 before entry into the target routine. The specified routine must exit with a RET instruction and should place a status value in R0 before returning.

The system loads R4 with the address of the Process Control Block (PCB).

System Service Descriptions

\$CMKRNL

CONDITION VALUES RETURNED

SS\$_NOPRIV

The process does not have the privilege to change mode to kernel.

All other values

All other values are returned from the executing routine.

\$CNTREG—Contract Program/Control Region

The Contract Program/Control Region service deletes a specified number of pages from the current end of the program or control region of a process's virtual address space. The deleted pages become inaccessible, and references to them cause access violations.

FORMAT **SY\$CNTREG** *pagcnt* ,*[retadr]* ,*[acmode]* ,*[region]*

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS

pagcnt

VMS Usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Number of pages to be deleted from the current end of the program or control region. The ***pagcnt*** argument is a longword specifying this number.

retadr

VMS Usage: **address_range**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Starting and ending pages of the deleted area. The ***retadr*** argument is the address of a two-longword array to receive the virtual addresses of the starting page and ending page of the deleted area.

acmode

VMS Usage: **access_mode**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Access mode of the owner of the pages to be deleted. The ***acmode*** argument is a longword containing the access mode. The \$PSLDEF macro defines the following symbols for the four access modes:

System Service Descriptions

\$CNTREG

Symbol	Access mode
PSL\$C_KERNEL	Kernel mode
PSL\$C_EXEC	Executive mode
PSL\$C_SUPER	Supervisor mode
PSL\$C_USER	User mode

The most privileged access mode used is the access mode of the caller.

region

VMS Usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Indicator specifying which region of memory (P0 or P1) is to be contracted. The **region** argument is a longword containing the indicator. A value of 0 (the default) indicates that the program region (P0 region) is to be contracted, and a value of 1 indicates that the control region (P1 region) is to be contracted.

DESCRIPTION

If an error occurs while deleting pages, the **retadr** argument, if specified, indicates the range of pages that were successfully deleted before the error occurred. If no pages were deleted, both longwords in **retadr** contain a -1.

The \$CNTREG service can delete pages only from the current end of the process's program or control region. To delete a specific range of pages in either region, use the Delete Virtual Address Space (\$DELTVA) service.

Note: Do not use the \$CNTREG, or \$CRETVA system services in conjunction with other user-written procedures and/or DIGITAL-supplied procedures (including run-time library procedures). These system services provide no means to communicate a change in virtual address space with other routines. DIGITAL recommends that you use either \$EXPREG or the Run-time Library Procedure Allocate Virtual Memory (LIB\$GET_VM) to get memory. You can find documentation on LIB\$GET_VM in the *VAX/VMS Run-Time Library Routines Reference Manual*. When using \$DELTVA, you should take care to only delete pages that you have specifically created.

CONDITION VALUES RETURNED

SS\$_NORMAL	Service successfully completed.
SS\$_ACCVIO	The retadr argument cannot be written by the caller.
SS\$_ILLPAGCNT	The specified page count was less than 1.
SS\$_PAGOWNVIO	A page in the specified range is owned by a more privileged access mode.

\$CRELNM—Create Logical Name

The Create Logical Name service creates a logical name and specifies its equivalence name(s).

FORMAT

SYSS\$CRELNM *[attr], tabnam , lognam , [acmode]
 , [itmlst]*

RETURNS

VMS Usage: cond_value
type: longword (unsigned)
access: write only
mechanism: by value

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS *attr*

VMS Usage: mask_longword
type: longword (unsigned)
access: read only
mechanism: by reference

Attributes to be associated with the logical name. The **attr** argument is the address of a longword bit mask specifying these attributes.

Each bit in the longword corresponds to an attribute and has a symbolic name. These symbolic names are defined by the `$LNMDEF` macro. To specify an attribute, specify its symbolic name or set its corresponding bit. The longword bit mask is the logical OR of all desired attributes. All undefined bits in the longword must be 0.

If this argument is not specified or is specified as 0 (no bits set), none of the attributes are associated with the logical name.

The following list describes each attribute:

Attribute	Description
LNMSM_CONFINE	If set, the logical name is not copied from the process to its spawned subprocesses. A subprocess is created through the DCL SPAWN command or the LIB\$SPAWN Run-Time Library routine. If the logical name is placed into a process-private table that has the CONFINE attribute, the CONFINE attribute is automatically associated with the logical name. This applies only to process-private logical names.
LNMSM_NO_ALIAS	If set, the logical name cannot be duplicated in this table at an outer access mode. If another logical name with the same name already exists in the table at an outer access mode, it is deleted.

System Service Descriptions

\$CRELNM

tabnam

VMS Usage: **logical_name**

type: **character-coded text string**

access: **read only**

mechanism: **by descriptor—fixed length string descriptor**

Name of the table in which to create the logical name. The **tabnam** argument is the address of a descriptor that points to the name of this table. This argument is required.

If **tabnam** is not the name of a logical name table, it is assumed to be a logical name and is translated iteratively until either the name of a logical name table is found or the number of translations allowed by the system has been performed. If **tabnam** translates to a list of logical name tables, the logical name is entered into the first table in the list.

SYSNAM or SYSPRV privilege is required to specify the system table, and GRPNAM or SYSPRV privilege is required to specify the group table.

SYSPRV privilege is required to specify the system directory table LNM\$SYSTEM_DIRECTORY.

lognam

VMS Usage: **logical_name**

type: **character-coded text string**

access: **read only**

mechanism: **by descriptor—fixed length string descriptor**

Name of the logical name to be created. The **lognam** argument is the address of a descriptor that points to the logical name string. Logical name strings of logical names created within either the system or process directory table must consist of alphanumeric characters, dollar signs, and underscores; the maximum length is 31 characters. The maximum length of logical name strings created within other tables is 255 characters with no restrictions on the types of characters that can be used. This argument is required.

acmode

VMS Usage: **access_mode**

type: **byte (unsigned)**

access: **read only**

mechanism: **by reference**

Access mode to be associated with the logical name. The **acmode** argument is the address of a byte that specifies the access mode.

The access mode associated with the logical name is determined by "maximizing" the access mode of the caller with the access mode specified by the **acmode** argument, which is to say that the less privileged of the two is used. Symbols for the four access modes are defined by the \$PSLDEF macro.

An access mode more privileged than that of the containing table may not be specified.

However, if the caller has SYSNAM privilege, then the specified access mode is associated with the logical name regardless of the access mode of the caller.

If this argument is omitted or is specified as 0, the access mode of the caller is associated with the logical name.

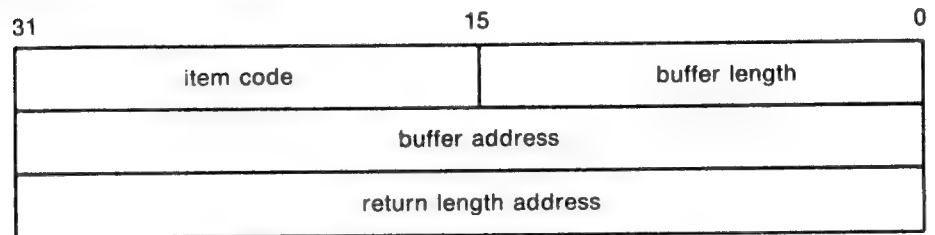
System Service Descriptions

\$CRELNM

itmlst

VMS Usage: **item_list_3**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Item list describing the equivalence name(s) to be defined for the logical name and information to be returned to the caller. The **itmlst** argument is the address of a list of item descriptors, each of which specifies information about an equivalence name. The list of item descriptors is terminated by a longword of 0. This argument is required. The following diagram depicts a single item descriptor:



ZK-1705-84

\$CRELNM Item Descriptor Fields

buffer length

A word specifying the number of bytes in the buffer pointed to by the **buffer address** field.

item code

A word that contains a symbolic code describing the nature of the information in the buffer or to be returned to the buffer pointed to by the **buffer address** field. The item codes are described under "**\$CRELNM Item Codes**."

buffer address

A longword containing the address of the buffer that receives or passes information.

return length address

A longword containing the address of a word that receives the actual length in bytes of the information returned by **\$CRELNM** in the buffer pointed to by the **buffer address** field. The **return length address** field is used only when the item code specified is **LNMS_TABLE**. Although this field is ignored for all other item codes, it must nevertheless be present as a placeholder in each item descriptor.

\$CRELNM Item Codes

LNMS_ATTRIBUTES

When **LNMS_ATTRIBUTES** is specified, the **buffer address** field of the item descriptor points to a longword bit mask that specifies the current translation attributes for the logical name. The current translation attributes are applied to all subsequently specified equivalence strings until another **LNMS_ATTRIBUTES** item descriptor is encountered. The symbolic names for these attributes are defined by the **\$LNMDEF** macro. The following provides the symbolic name and description of each attribute.

System Service Descriptions

\$CRELNM

Attribute	Description
LNMSM_CONCEALED	If set, RMS interprets the equivalence name as a device name or logical name with the LNMSM_CONCEALED attribute.
LNMSM_TERMINAL	If set, further iterative logical name translation on the equivalence name is not to be performed.

LNMS_CHAIN

When LNMS_CHAIN is specified, the **buffer address** field of the item descriptor points to another item list that \$CRELNM is to process immediately after it has processed the current item list.

If the LNMS_CHAIN item code is specified, it must be the last item code in the current item list.

LNMS_STRING

When LNMS_STRING is specified, the **buffer address** field of the item descriptor points to a buffer containing a user-specified equivalence name for the logical name. The maximum length of the equivalence string is 255 characters.

When \$CRELNM encounters an item descriptor with the item code LNMS_STRING, it creates an equivalence name entry for the logical name using the most recently specified values for LNMS_ATTRIBUTES. The equivalence name entry includes the following information:

- The name specified by LNMS_STRING.
- The next available index value. Each equivalence is assigned a unique value from 0 to 127.
- The attributes specified by the most recently encountered item descriptor with item code LNMS_ATTRIBUTES (if these are present in the item list).

Therefore, you should construct the item list so that the LNMS_ATTRIBUTES item codes immediately precede the LNMS_STRING item code(s) to which they apply.

LNMS_TABLE

When LNMS_TABLE is specified, the **buffer address** field of the item descriptor points to a buffer in which \$CRELNM writes the name of the logical name table in which it entered the logical name. The **return length address** field points to a word that contains a buffer that specifies the length in bytes of the information returned by \$CRELNM. The maximum length of the name of a logical name table is 31 characters.

This item code may appear anywhere in the item list.

DESCRIPTION

The calling process must have the following:

- Write access to shareable tables to create logical names in those tables
- SYSNAM privilege to create executive or kernel mode logical names
- GRPNAM or SYSPRV privilege to enter a logical name into the group logical name table

System Service Descriptions

\$CRELNM

- SYSNAM or SYSPRV privilege to enter a logical name into the system logical name table

CONDITION VALUES RETURNED

SS\$_NORMAL	Successful completion; the logical name has been created.
SS\$_SUPERSEDE	Successful completion; the logical name has been created and a previously existing logical name with the same name has been deleted.
SS\$_BUFFEROVF	Successful completion; the buffer length field in an item descriptor specified an insufficient value, so the buffer was not large enough to hold the requested data.
SS\$_ACCVIO	The service cannot access the location(s) specified by one or more arguments.
SS\$_BADPARAM	One or more arguments has an invalid value, or a logical name table name or logical name was not specified.
SS\$_DUPLNAM	An attempt was made to create a logical name with the same name as an already existing logical name, and the existing logical name was created at a more privileged access mode and with the LNM\$_NO_ALIAS attribute.
SS\$_EXLNMQUOTA	Insufficient quota associated with the specified logical name table for the creation of the logical name.
SS\$_INSFMEM	Dynamic memory is insufficient for the creation of the logical name.
SS\$_JVLOGNAM	The tabnam argument, lognam argument, or the equivalence string specifies a string whose length is not in the required range of 1 through 255 characters. The lognam argument specifies a string whose length is not in the required range of 1 to 31 characters for directory table entries.
SS\$_JVLOGTAB	The tabnam argument does not specify a logical name table.
SS\$_NOLOGTAB	Either the specified logical name table does not exist or the logical name translation of the table name exceeded the allowable depth of 10 translations.
SS\$_NOPRIV	The caller lacks the necessary privilege to create the logical name.

\$CRELNT

The **Create Logical Name Table** service creates a process-private or shareable logical name table.

RETURNS

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can returned by this service are listed under "CONDITION VALUES RETURNED."

The following list describes each attribute:

System Service Descriptions

\$CRELNT

Attribute	Description
LNMSM_CONFINE (Cont.)	<p>The state of this bit is also propagated from the parent table to the newly created table and can be overridden only if the parent table does not have the bit set. Thus, if the parent table has the LNMSM_CONFINE attribute, the newly created table will also have it, no matter what is specified in the attr argument. On the other hand, if the parent table does not have the LNMSM_CONFINE attribute, the newly created table can be given this attribute through the attr argument.</p> <p>The process-private directory table LNM\$PROCESS_DIRECTORY does not have the LNMSM_CONFINE attribute.</p>
LNMSM_CREATE_IF	<p>If set, a new logical name table is created only if the specified table name is not already entered at the specified access mode in the appropriate directory table. If the table name exists, a new table is not created and no modification is made to the existing table name. This holds true even if the existing name has differing attributes or quota values, or even if it is not the name of a logical name table.</p> <p>If LNMSM_CREATE_IF is not set, the new logical name table will supersede any existing table name with the same access mode within the appropriate directory table. Setting this attribute is useful when two or more people want to create and use the same table but do not want to synchronize its creation.</p>
LNMSM_NO_ALIAS	<p>If set, the name of the logical name table cannot be duplicated at an outer access mode within the appropriate directory table. If this name already exists at an outer access mode, it is deleted.</p>

resnam

VMS Usage: **logical_name**
 type: **character-coded text string**
 access: **write only**
 mechanism: **by descriptor—fixed length string descriptor**

Name of the newly created logical name table, returned by \$CRELNT. The **resnam** argument is the address of a descriptor pointing to this name. The name is a character string whose maximum length is 31 characters.

reslen

VMS Usage: **word_unsigned**
 type: **word (unsigned)**
 access: **write only**
 mechanism: **by reference**

Length in bytes of the name of the newly created logical name table, returned by \$CRELNT. The **reslen** argument is the address of a word to receive this length.

System Service Descriptions

\$CRELNT

quota

VMS Usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Maximum number of bytes of memory to be allocated for logical names contained in this logical name table. The **quota** argument is the address of a longword specifying this value.

If no quota value is specified, the logical name table has an infinite quota. Note that a shareable table created with infinite quota permits users with write access to that table to consume system dynamic memory without limit.

promsk

VMS Usage: **file_protection**
type: **word (unsigned)**
access: **read only**
mechanism: **by reference**

Protection mask to be associated with the newly created shareable logical name table. The **promsk** argument is the address of a word that contains a value that represents four 4-bit fields, where each field describes the type of access allowed for system, owner, group, and world users.

WORLD				GROUP				OWNER				SYSTEM			
D	E	W	R	D	E	W	R	D	E	W	R	D	E	W	R
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

ZK-1706-84

Each field consists of four bits specifying protection for the logical name table. The "E" protection bit is reserved for DIGITAL use. The remaining bits in the protection mask are as follows:

- Read privileges allow access to names in the logical name table.
- Write privileges allow creation and deletion of names within the logical name table.
- Delete privileges allow deletion of the logical name table.

Note: The "E" protection bit is reserved for DIGITAL use.

If a bit is clear, access is granted. If the mask is omitted, complete access is granted to system and owner, and no access is granted to world and group.

tabnam

VMS Usage: **logical_name**
type: **character-coded text string**
access: **read only**
mechanism: **by descriptor—fixed length string descriptor**

The name of the new logical name table. The **tabnam** argument is the address of a character string descriptor pointing to this name string. Table names are contained in either the process or system directory table (LNM\$PROCESS_DIRECTORY or LNM\$SYSTEM_DIRECTORY). Therefore

System Service Descriptions

\$CRELNT

table names must consist of alphanumeric characters, dollar signs, and underscores; the maximum length is 31 characters.

If this argument is not specified, a default name in the format LNM\$xxxx is used, where xxxx is a unique hexadecimal number.

SYSRV privilege is required in order to be able to specify the name of a shareable logical name table.

partab

VMS Usage: **char_string**

type: **character-coded text string**

access: **read only**

mechanism: **by descriptor—fixed length string descriptor**

Parent table name. The **partab** argument is the address of a character string descriptor pointing to this name string. The only valid values for this argument are the names of the directory tables LNM\$PROCESS_DIRECTORY or LNM\$SYSTEM_DIRECTORY. SYSRV privilege is required to specify the system directory table LNM\$SYSTEM_DIRECTORY. This argument is required.

acmode

VMS Usage: **access_mode**

type: **byte (unsigned)**

access: **read only**

mechanism: **by reference**

Access mode to be associated with the newly created logical name table. The **acmode** argument is the address of a byte containing this access mode. The \$PSLDEF macro defines symbolic names for the four access modes.

If the **acmode** argument is not specified or is specified as 0, the access mode of the caller is associated with the newly created logical name table.

The access mode associated with the logical name table is determined by "maximizing" the access mode of the caller with the access mode specified by the **acmode**. The less privileged of the two access modes is used.

However, if the caller has SYSNAM privilege, then the specified access mode is associated with the logical name table, regardless of the access mode of the caller.

Access modes associated with logical name tables govern logical name table processing and provide a protection mechanism that prevents the deletion of inner access mode logical name tables by nonprivileged users.

An access mode more privileged than that of the parent table may not be specified.

A logical name table with supervisor mode access may contain supervisor mode and user mode logical names and may be a parent to supervisor mode and user mode logical name tables, but may not contain executive or kernel mode logical names or be a parent to executive or kernel mode logical name tables.

SYSNAM privilege is required to specify executive or kernel mode access for a logical name table.

System Service Descriptions

\$CRELNT

DESCRIPTION Depending on the operation, use of \$CRELNT may require the calling process to have certain privilege:

- SYSPRV privilege is required to create a shareable table
- SYSNAM privilege is required to create a table at an access mode more privileged than that of the calling process

\$CRELNT uses the following system resources:

- System paged dynamic memory is required to create a shareable logical name table
- Process dynamic memory is required to create a process-private logical name table

The parent table governs whether the new table is process-private or shareable. If the parent table is process-private, so is the new table; if the parent table is shareable, so is the new table.

CONDITION VALUES RETURNED

SS\$_NORMAL	Successful completion; the logical name table already exists.
SS\$_LNMCREATED	Successful completion; the logical name table was created.
SS\$_SUPERSEDE	Successful completion; the logical name table was created and its logical name superseded already existing logical name(s) in the directory table.
SS\$_ACCVIO	The service cannot access the location(s) specified by one or more arguments.
SS\$_BADPARAM	One or more arguments has an invalid value, or a parent logical name table was not specified.
SS\$_DUPLNAM	An attempt was made to create a logical name table with the same name as an already existing name within the appropriate directory table and the existing name was created at a more privileged access mode and with the LNM\$_NO_ALIAS attribute.
SS\$_EXLNMQUOTA	The parent table has insufficient quota for the creation of the new table.
SS\$_INSFMEM	Dynamic memory is insufficient for the creation of the table.
SS\$_JVLOGNAM	The partab argument specifies a string whose length is not within the required range of 1 to 31 characters.
SS\$_JVLOGTAB	The tabnam argument is not alphanumeric or specifies a string whose length is not within the required range of 1 to 31 characters.
SS\$_NOLOGTAB	The parent logical name table does not exist.
SS\$_NOPRIV	The caller lacks the necessary privilege to create the table.

System Service Descriptions

\$CRELNT

SS\$_PARENT_DEL

Creation of the new table would have resulted in the deletion of the parent table.

SS\$_RESULTOVF

The table name buffer is not large enough to contain the name of the new table.

System Service Descriptions

\$CREMBX

\$CREMBX—Create Mailbox and Assign Channel

The Create Mailbox and Assign Channel service creates a virtual mailbox device named MBAn: and assigns an I/O channel to it. The system provides the unit number, *n*, when it creates the mailbox. If a mailbox with the specified name already exists, the \$CREMBX service assigns a channel to the existing mailbox.

FORMAT	SY\$CREMBX [<i>prmflg</i>] , <i>chan</i> ,[<i>maxmsg</i>] ,[<i>bufquo</i>] ,[<i>promsk</i>] ,[<i>acmode</i>] ,[<i>lognam</i>]
---------------	--

RETURNS	<p>VMS Usage: cond_value type: longword (unsigned) access: write only mechanism: by value</p> <p>Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."</p>
----------------	---

ARGUMENTS	<p><i>prmflg</i> VMS Usage: boolean type: byte (unsigned) access: read only mechanism: by value</p> <p>Indicator specifying whether the created mailbox is to be permanent or temporary. The <i>prmflg</i> argument is a byte value. A value of 1 specifies a permanent mailbox; a value of 0, which is the default, specifies a temporary mailbox. Any other values result in an error.</p>
------------------	--

chan
VMS Usage: **channel**
type: **word (unsigned)**
access: **write only**
mechanism: **by reference**

Channel number assigned by \$CREMBX to the mailbox. The ***chan*** argument is the address of a word into which \$CREMBX writes the channel number.

maxmsg
VMS Usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Maximum size (in bytes) of a message that can be sent to the mailbox. The ***maxmsg*** argument is a longword value containing this size. If ***maxmsg*** is not specified or is specified as 0, VAX/VMS provides a default value.

System Service Descriptions

\$CREMBX

bufquo

VMS Usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Number of bytes of system dynamic memory that can be used to buffer messages sent to the mailbox. The **bufquo** argument is a longword value containing this number. If **bufquo** is not specified or is specified as 0, VAX/VMS provides a default value.

For a temporary mailbox, this value must be less than or equal to the process buffer quota.

promsk

VMS Usage: **file_protection**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Protection mask to be associated with the created mailbox. The **promsk** argument is a longword value that is the combined value of the bits set in the protection mask. Cleared bits grant access and set bits deny access to each of the four classes of user: world, group, owner, and system. The following diagram depicts these protection bits.

WORLD				GROUP				OWNER				SYSTEM			
L	P	W	R	L	P	W	R	L	P	W	R	L	P	W	R
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

ZK-1707-84

If the **promsk** argument is not specified or is specified as 0, read, write, physical, and logical access are granted to all users.

The physical access bit is ignored for all categories of user. The logical access bit must be clear for all categories of user because logical access is required to read or write to a mailbox; thus, setting or clearing the read and write access bits is meaningless unless the logical access bit is also cleared.

Logical access also allows the user to queue read or write attention ASTs.

acmode

VMS Usage: **access_mode**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Access mode to be associated with the channel to which the mailbox is assigned. The **acmode** argument is a longword containing the access mode. The \$PSLDEF macro defines the following symbols for the four access modes:

System Service Descriptions

\$CREMBX

Symbol	Access mode
PSL\$C_KERNEL	Kernel mode
PSL\$C_EXEC	Executive mode
PSL\$C_SUPER	Supervisor mode
PSL\$C_USER	User mode

The most privileged access mode used is the access mode of the caller.

lognam

VMS Usage: **logical_name**

type: **character-coded text string**

access: **read only**

mechanism: **by descriptor—fixed length string descriptor**

Logical name to be assigned to the mailbox. The **lognam** argument is the address of a character string descriptor pointing to the logical name string.

The equivalence name for the mailbox is MBAn:. The equivalence name is marked with the terminal attribute. Processes can use the logical name to assign other I/O channels to the mailbox.

For permanent mailboxes, the \$CREMBX service enters the specified logical name, if any, in the LNM\$PERMANENT_MAILBOX logical name table; for temporary mailboxes, into the LNM\$TEMPORARY_MAILBOX logical name table.

DESCRIPTION

Depending on the operation, use of \$CREMBX may require the calling process to have certain privilege:

- TMPMBX privilege is required to create a temporary mailbox
- PRMMBX privilege is required to create a permanent mailbox
- SHMEM privilege is required to create a mailbox in memory shared by multiple processors
- SYSNAM privilege is required to place a logical name for a mailbox in the system logical name table
- GRPNAM privilege is required to place a logical name for a mailbox in the group logical name table

\$CREMBX uses the following system resources:

- System dynamic memory is required for the allocation of a device database for the mailbox and for an entry in the logical name table (if a logical name is specified)
- When a temporary mailbox is created, the process's buffered I/O byte count (BYTLM) quota is reduced by the amount specified in the BUFQUO argument. The size of the mailbox unit control block and the logical name (if specified) are also subtracted from the quota. The quota is returned to the process when the mailbox is deleted.
- The creation of a mailbox in shared memory requires MAILBOX PORT quota. This quota is acquired by means of the SYSGEN command SHARE and is returned when the mailbox is deleted.

System Service Descriptions

\$CREMBX

After a mailbox is created, the creating process and other processes can assign additional channels to it by calling the Assign I/O Channel (\$ASSIGN) or Create Mailbox (\$CREMBX) services. If the mailbox already exists, the \$CREMBX service simply assigns a channel to that mailbox; in this way, cooperating processes need not consider which process must execute first to create the mailbox. The system maintains a reference count of the number of channels assigned to a mailbox, and this count is decreased whenever a channel is deassigned with the Deassign I/O Channel (\$DASSGN) service or when the image that assigned the channel exits.

A temporary mailbox is deleted when there are no more channels assigned to it. A permanent mailbox must be explicitly marked for deletion with the Delete Mailbox (\$DELMBX) service; its actual deletion occurs when no more channels are assigned to it.

A mailbox is treated as a shareable device; it cannot, however, be mounted or allocated.

Mailboxes are assigned sequentially increasing unit numbers (from 1 to a maximum of 9999) as they are created. When all unit numbers have been used, the system starts numbering again at unit 1.

A process can obtain the unit number of the created mailbox by calling the Get Device/Volume Information (\$GETDVI) service.

Default values for the maximum message size and the buffer quota (an appropriate multiple of the message size) are determined for a specific system during system generation. The SYSGEN parameter DEFMBXMXMSG determines the maximum message size; the SYSGEN parameter DEFMBXBUFQUO determines the buffer quota. For termination mailboxes, the maximum message size must be at least as large as the termination message (currently 84 bytes).

When you specify a logical name for a temporary mailbox, the \$CREMBX service enters the name into the LNM\$TEMPORARY_MAILBOX logical name table.

Normally, LNM\$TEMPORARY_MAILBOX specifies LNM\$JOB, the job-wide logical name table; thus, only processes in the same job as the process that first creates the mailbox can use the logical name to access the temporary mailbox. If you want to use the temporary mailbox to enable communication between processes in different jobs, you must redefine LNM\$TEMPORARY_MAILBOX in the process logical name directory table (LNM\$PROCESS_DIRECTORY), to specify a logical name table that those processes can access.

For instance, if you want to use the mailbox as a communication device for processes in the same group, you must redefine LNM\$TEMPORARY_MAILBOX to specify LNM\$GROUP, the group logical name table. The following DCL command assigns temporary mailbox logical names to the group logical name table:

```
$DEFINE/TABLE=LNM$PROCESS_DIRECTORY -  
LNM$TEMPORARY_MAILBOX LNM$GROUP
```

When you specify a logical name for a permanent mailbox, the system enters the name in the logical name table specified by the logical name table name LNM\$PERMANENT_MAILBOX. LNM\$PERMANENT_MAILBOX normally specifies LNM\$SYSTEM, the system logical name table. If you want the logical name that you specify for the mailbox to be entered in a logical name table other than the system logical name table, redefine LNM\$PERMANENT_MAILBOX to specify the desired table. For more information on logical name tables see Section 6.

System Service Descriptions

SCREMBX

If you redefine either LNM\$TEMPORARY_MAILBOX or LNM\$PERMANENT_MAILBOX, be sure that the name of the new table appears in the logical name table LNM\$FILE_DEV. RMS and the system I/O services use LNM\$FILE_DEV to translate I/O device names. If the logical name table specified by either LNM\$TEMPORARY_MAILBOX or LNM\$PERMANENT_MAILBOX does not appear in LNM\$FILE_DEV, the system will be unable to translate the logical name of your mailbox and therefore, will be unable to access your mailbox as an I/O device.

CONDITION VALUES RETURNED

SS\$_NORMAL	Service successfully completed.
SS\$_ACCVIO	The logical name string or string descriptor cannot be read by the caller, or the channel number cannot be written by the caller.
SS\$_BADPARAM	The bufquo argument specified a value greater than approximately 65355, which is 65535 minus the size of a mailbox unit control block (UCB).
SS\$_EXBYTLM	The process has insufficient buffer I/O byte count (BYTLM) quota to allocate the mailbox UCB or to satisfy buffer requirements.
SS\$_EXPORTQUOTA	The process has exceeded the number of mailboxes that processes on this port of the multiport (shared) memory can create.
SS\$_INSFMEM	Insufficient system dynamic memory is available to complete the service.
SS\$_INTERLOCK	The bit map lock for allocating mailboxes from the specified shared memory is locked by another process.
SS\$_IVLOGNAM	The logical name string has a length of 0 or has more than 255 characters.
SS\$_IVSTSFLG	Undefined bit set in the prmfllg argument; this argument may have a value of 1 or 0.
SS\$_NOIOCHAN	No I/O channel is available for assignment.
SS\$_NOPRIV	The process does not have the privilege to create a temporary mailbox, a permanent mailbox, a mailbox in memory that is shared by multiple processors, or a logical name.
SS\$_NOSHMBLOCK	No shared memory mailbox control block is available to use to create a new mailbox.
SS\$_OPINCOMPL	A duplicate unit number was encountered while linking a shared memory mailbox UCB. If this condition value is returned, submit an SPR to DIGITAL.
SS\$_SHMNOTCNCT	The shared memory named in the lognam argument is not known to the system. This error can be caused by a spelling error in the string, an improperly assigned logical name, or the failure to identify the memory as shared at system generation time.
SS\$_TOOMANYLNAM	Logical name translation of the string named in the lognam argument exceeded the allowed depth.

\$CREPRC—Create Process

The Create Process service creates a subprocess or detached process on behalf of the calling process.

FORMAT

SYSCREPRC *[pidadr],[image],[input],[output]
[error],[prvadr],[quota],[prcnam]
[baspri],[uic],[mbxunt],[stsflg]*

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS

pidadr

```

VMS Usage:  process_id
type:       longword (unsigned)
access:     write only
mechanism:  by reference

```

Process identification (PID) of the newly created process. The **pidadr** argument is the address of a longword into which \$CREPRC writes the PID.

image

VMS Usage: **logical_name**
type: **character-coded text string**
access: **read only**
mechanism: **by descriptor—fixed length string descriptor**

Name of the image to be activated in the newly created process. The **image** argument is the address of a character string descriptor pointing to the file specification of the image.

The image name can have a maximum of 63 characters. If the image name contains a logical name, the equivalence name must be in a logical name table that can be accessed by the created process.

input

VMS Usage: **logical_name**
type: **character-coded text string**
access: **read only**
mechanism: **by descriptor—fixed length string descriptor**

Equivalence name to be associated with the logical name SYS\$INPUT in the logical name table of the created process. The **input** argument is the address of a character string descriptor pointing to the equivalence name string.

System Service Descriptions

\$CREPRC

output

VMS Usage: **logical_name**

type: **character-coded text string**

access: **read only**

mechanism: **by descriptor—fixed length string descriptor**

Equivalence name to be associated with the logical name SYS\$OUTPUT in the logical name table of the created process. The **output** argument is the address of a character string descriptor pointing to the equivalence name string.

error

VMS Usage: **logical_name**

type: **character-coded text string**

access: **read only**

mechanism: **by descriptor—fixed length string descriptor**

Equivalence name to be associated with the logical name SYS\$error in the logical name table of the created process. The **error** argument is the address of a character string descriptor pointing to the equivalence name string.

Note that the **error** argument is ignored if the **image** argument specifies SYS\$SYSTEM:LOGINOUT.EXE; in this case, SYS\$error will point to SYS\$OUTPUT.

prvadr

VMS Usage: **mask_privileges**

type: **quadword (unsigned)**

access: **read only**

mechanism: **by reference**

Privileges to be given to the created process. The **prvadr** argument is the address of a quadword bit vector wherein each bit corresponds to a privilege; setting a bit gives the privilege.

Each bit has a symbolic name; these names are defined by the \$PRVDEF macro. The bit vector is formed by specifying the symbolic name of each desired privilege in a logical OR operation. Table SYS-5 gives the symbolic name and description of each privilege:

Table SYS-5 User Privileges

Privilege	Symbolic name	Description
ALLSPOOL	PRV\$V_ALLSPOOL	Allocate a spooled device
BUGCHK	PRV\$V_BUGCHK	Make bugcheck error log entries
BYPASS	PRV\$V_BYPASS	Bypass UIC-based protection
CMEXEC	PRV\$V_CMEXEC	Change mode to executive
CMKRNL	PRV\$V_CMKRNL	Change mode to kernel
DETACH	PRV\$V_DETACH	Create detached processes
DIAGNOSE	PRV\$V_DIAGNOSE	May diagnose devices
DOWNGRADE	PRV\$V_DOWNGRADE	May downgrade classification
EXQUOTA	PRV\$V_EXQUOTA	May exceed quotas
GROUP	PRV\$V_GROUP	Group process control

System Service Descriptions

\$CREPRC

Table SYS-5 (Cont.) User Privileges

Privilege	Symbolic name	Description
GRPNAM	PRV\$V_GRPNAM	Place name in group logical name table
GRPPRV	PRV\$V_GRPPRV	Group access via system protection field
LOG_IO	PRV\$V_LOG_IO	Perform logical I/O operations
MOUNT	PRV\$V_MOUNT	Issue mount volume QIO
NETMBX	PRV\$V_NETMBX	Create a network device
ACNT	PRV\$V_NOACNT	Create processes for which no accounting is done
OPER	PRV\$V_OPER	All operator privileges
PFNMAP	PRV\$V_PFNMAP	Map to section by physical page frame number
PHY_IO	PRV\$V_PHY_IO	Perform physical I/O operations
PRMCEB	PRV\$V_PRMCEB	Create permanent common event flag clusters
PRMGBL	PRV\$V_PRMGBL	Create permanent global sections
PRMJNL	PRV\$V_PRMJNL	May create permanent journals
PRMMBX	PRV\$V_PRMMBX	Create permanent mailboxes
PSWAPM	PRV\$V_PSWAPM	Change process swap mode
READALL	PRV\$V_READALL	Possess read access to everything
SECURITY	PRV\$V_SECURITY	May perform security functions
ALTPRI	PRV\$V_SETPRI	Set (alter) any process priority
SETPRV	PRV\$V_SETPRV	Set any process privileges
SHARE	PRV\$V_SHARE	May assign a channel to a non-shared device
SHMEM	PRV\$V_SHMEM	Allocate structures in memory shared by multiple processors
SYSGBL	PRV\$V_SYSGBL	Create system global sections
SYSLCK	PRV\$V_SYSLCK	Queue system-wide locks
SYSNAM	PRV\$V_SYSNAM	Place name in system logical name table
SYSPRV	PRV\$V_SYSPRV	Access files and other resources as if you have a system UIC
TMPJNL	PRV\$V_TMPJNL	May create temporary journals
TMPMBX	PRV\$V_TMPMBX	Create temporary mailboxes
UPGRADE	PRV\$V_UPGRADE	May upgrade classification
VOLPRO	PRV\$V_VOLPRO	Override volume protection
WORLD	PRV\$V_WORLD	World process control

Note that the names of the privilege bits PRV\$V_NOACNT and PRV\$V_SETPRI correspond to the names of the DCL privileges ACNT and ALTPRI, yet have different names.

System Service Descriptions

\$CREPRC

If this quota is not specified and the created process is a detached process, the detached process receives a default value of 0, that is, unlimited CPU time.

If this quota is not specified and the created process is a subprocess, the subprocess receives half the CPU time limit quota of the creating process.

If this quota is specified as 0, the created process has unlimited CPU time providing that the creating process also has unlimited CPU time. If, however, the creating process does not have unlimited CPU time, the created process receives half the CPU time limit quota of the creating process.

The CPU time limit quota is a consumable quota; that is, the amount of CPU time used by the created process is not returned to the creating process when the created process is deleted.

Minimum: PQL_MCPULM

Default: PQL_DCPULM

Deductible

PQL\$_DIOLM

Direct I/O quota. This quota limits the number of outstanding direct I/O operations. A direct I/O operation is one for which the system locks the pages containing the associated I/O buffer in memory for the duration of the I/O operation.

Minimum: PQL_MDIOLM

Default: PQL_DDIOLM

Nondeductible

PQL\$_ENQLM

Lock request quota. This quota limits the number of lock requests that a process can queue.

Minimum: PQL_MENQLM

Default: PQL_DENQLM

Pooled

PQL\$_FILLM

Open file quota. This quota limits the number of files that a process can have open at one time.

Minimum: PQL_MFILLM

Default: PQL_DFILLM

Pooled

PQL\$_JTQUOTA

Job table quota. This quota limits the number of bytes of system paged pool used for the job logical name table. This item is ignored if the process being created is a subprocess.

Minimum: PQL_MJTQUOA

Default: PQL_DJTQUOTA

Deductible

System Service Descriptions

\$CREPRC

PQL\$_PGFLQUOTA

Paging file quota. This quota limits the number of pages that can be used to provide secondary storage in the paging file for a process's execution.

Minimum: PQL_MPGFLQUOTA

Default: PQL_DPGFLQUOTA

Pooled

PQL\$_PRCLM

Subprocess quota. This quota limits the number of subprocesses a process can create.

Minimum: PQL_MPRCLM

Default: PQL_DPRCLM

Pooled

PQL\$_TQELM

Timer queue entry quota. This quota limits both the number of timer queue requests a process can have outstanding and the creation of temporary common event flag clusters.

Minimum: PQL_MTQELM

Default: PQL_DTQELM

Pooled

PQL\$_WSDEFAULT

Default working set size. This quota defines the number of pages in the default working set for any image executed by the process. The maximum size that can be specified for this quota is determined by the working set size quota.

Minimum: PQL_MWSDEFAULT

Default: PQL_DWSDEFAULT

Nondeductible

PQL\$_WSEXTENT

Working set expansion quota. This quota limits the maximum size to which an image can expand its working set size with the Adjust Working Set Limit (\$ADJWSL) system service.

Minimum: PQL_MWSEXTENT

Default: PQL_DWSEXTENT

Nondeductible

PQL\$_WSQUOTA

Working set size quota. This quota limits the maximum size to which an image can lock pages in its working set with the Lock Pages in Memory (\$LCKPAG) system service.

Minimum: PQL_MWSQUOTA

Default: PQL_DWSQUOTA

Nondeductible

System Service Descriptions

\$CREPRC

Use of the Quota List

Values specified in the quota list are not necessarily the quotas that will actually be assigned to the created process. The \$CREPRC service performs the following steps to determine the quota values that will be assigned:

- 1 It constructs a default quota list for the process being created, assigning it the default values for all quotas. Default values are SYSGEN parameters and so may vary from system to system.
- 2 It reads the specified quota list, if any, and updates the corresponding items in the default list. If the quota list contains multiple entries for a quota, only the last specification is used.
- 3 For each item in the updated quota list, it compares the quota value with the minimum value required (also a SYSGEN parameter) and uses the larger value. Then:
 - If a subprocess is being created or a detached process is being created and the creator does not have DETACH privilege, the resulting value is compared with the current value of the corresponding quota of the creator and the lesser value is used.

Then, if the quota is a deductible quota, that value is deducted from the creator's quota, and a check is performed to ensure that the creator will still have at least the minimum quota required. If not, the condition value SS\$_EXQUOTA is returned and the subprocess or detached process is not created.

Pooled quota values are ignored.

- If a detached process is being created and the creator has DETACH privilege, the resulting value is not compared with the current value of the corresponding quota of the creator and the resulting value is not deducted from the creator's quota. The \$CREPRC service does not check that a specified quota value exceeds the maximum allowed by the system.

prcnam

VMS Usage: **process_name**

type: **character-coded text string**

access: **read only**

mechanism: **by descriptor—fixed length string descriptor**

Process name to be assigned to the created process. The **prcnam** is the address of a character string descriptor pointing to a 1- to 15-character process name string.

If a subprocess is being created, the process name is implicitly qualified by the UIC group number of the creating process. If a detached process is being created, the process name is qualified by the group number specified in the **uic** argument.

System Service Descriptions

SCREPRC

baspri

VMS Usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Base priority to be assigned to the created process. The **baspri** argument is a longword value in the range 0 to 31, where 31 is the highest possible priority and 0 is the lowest. Normal priorities are in the range 0 through 15, and real-time priorities are in the range 16 through 31.

ALTPRI privilege is required to set a priority higher than one's own. If the caller does not have this privilege, the specified base priority is compared with the caller's priority and the lower of the two values is used.

uic

VMS Usage: **uic**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

User identification code (UIC) to be assigned to the created process. The **uic** argument is a longword value containing the UIC.

If **uic** is not specified or is specified as 0 (the default), \$CREPRC creates a process and assigns it the UIC of the creating process.

If **uic** specifies a nonzero value, \$CREPRC creates a detached process. The specified value is interpreted as a 32-bit octal number, with two 16-bit fields

bits 0-15—member number
bits 16-31—group number

DETACH privilege is required to create a detached process with a UIC that is different than the UIC of the creating process.

mbxunt

VMS Usage: **word_unsigned**
type: **word (unsigned)**
access: **read only**
mechanism: **by value**

Unit number of a mailbox to receive a termination message when the created process is deleted. The **mbxunt** argument is a word containing this number.

If **mbxunt** is not specified or is specified as 0 (the default), VAX/VMS sends no termination message when it deletes the process.

The Get Device/Volume Information (\$GETDVI) service must be used to obtain the unit number of the mailbox.

If **mbxunt** is specified, the mailbox is not used until the created process actually terminates. At that time, the \$ASSIGN service is issued for the mailbox in the context of the terminating process and an accounting message is sent to the mailbox. If the mailbox no longer exists, cannot be assigned, or is full, the error is treated as if no mailbox had been specified.

The accounting message is sent before process run-down is initiated but after the process name has been set to null. Thus, a significant interval of time can occur between the sending of the accounting message and the final deletion of the process.

System Service Descriptions

\$CREPRC

To receive the accounting message, the caller must issue a read to the mailbox. When the I/O completes, the second longword of the I/O status block, if one is specified, contains the process identification of the deleted process.

Symbolic names for offsets of fields within the accounting message are defined in the \$ACCDEF macro. The offsets, their symbolic names, and the contents of each field are listed below. Unless stated otherwise, the length of the field is four bytes.

Offset	Name	Contents
0	ACC\$W_MSGTYP	MSG\$_DELPROC (2 bytes)
2		Not used (2 bytes)
4	ACC\$L_FINALSTS	Exit status code
8	ACC\$L_PID	Process identification
12		Not used (4 bytes)
16	ACC\$Q_TERMTIME	Current time in system format at process termination (8bytes)
24	ACC\$T_ACCOUNT	Account name for process, blank filled (8 bytes)
32	ACC\$T_USERNAME	User name, blank filled (12 bytes)
44	ACC\$L_CPUTIM	CPU time used by the process, in 10-millisecond units
48	ACC\$L_PAGEFLT	Number of page faults incurred by the process
52	ACC\$L_PGFLPEAK	Peak paging file usage
56	ACC\$L_WSPEAK	Peak working set size
60	ACC\$L_BIOCNT	Count of buffered I/O operations performed by the process
64	ACC\$L_DIOCNT	Count of direct I/O operations performed by the process
68	ACC\$L_VOLUMES	Count of volumes mounted by the process
72	ACC\$Q_LOGIN	Time, in system format, that process logged in (8 bytes)
80	ACC\$L_OWNER	Process identification of owner

The length of the termination message is equated to the constant ACC\$K_TERMLEN.

System Service Descriptions

\$CREPRC

stsflg

VMS Usage: **mask_longword**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Options selected for the created process. The **stsflg** argument is a longword bit vector wherein a bit corresponds to an option. Only bits 0 to 10 are used; bits 11 to 31 are reserved and must be 0.

Each option (bit) has a symbolic name, which is defined by the \$PRCDEF macro. The **stsflg** argument is constructed by performing a logical OR operation using the symbolic names of each desired option. The following list describes the symbolic name of each option:

Symbolic Name	Description
PRC\$_SSRWAIT	Disable resource wait mode.
PRC\$_SSFEXCU	Enable system service failure exception mode.
PRC\$_PSWAPM	Inhibit process swapping. PSWAPM privilege is required.
PRC\$_NOACNT	Do not perform accounting. NOACNT privilege is required.
PRC\$_BATCH	Create a batch process. DETACH privilege is required.
PRC\$_HIBER	Force process to hibernate before it executes the image.
PRC\$_IMGDMP	Enable image dump facility. If an image terminates due to an unhandled condition, the image dump facility writes the contents of the address space to a file in your current default directory. The file name will be the same as the image name which terminated. The file type will be DMP.
PRC\$_NOUAF	Do not check authorization file if the process is detached and the image is LOGINOUT.EXE. This option should not be specified if a subprocess is being created. In previous versions of VAX/VMS, the symbolic name of this option was PRC\$_LOGIN. The symbolic name has been changed to more accurately denote the effect of setting this bit. For compatibility with existing user programs, users can still specify this bit as PRC\$_LOGIN.
PRC\$_NETWRK	Create a process that is a network connect object. DETACH privilege required.
PRC\$_DISAWS	Disable system initiated working set adjustment.
PRC\$_DETACH	Create a detached process.

System Service Descriptions

\$CREPRC

Symbolic Name	Description
PRC\$_INTER	Create an interactive process. This option is meaningful only if the image argument specifies SYS\$SYSTEM:LOGINOUT.EXE. The purpose of this option is to provide users with information about the process. When this option is specified it identifies the process as one that is in communication with a person, an interactive process. For example, if you make an inquiry, using the DCL lexical function F\$MODE, about a process that has specified the PRC\$_INTER option, F\$MODE returns the value "INTERACTIVE".
PRC\$_NOPASSWORD	Do not display the USERNAME: and PASSWORD: prompts if the process is interactive and detached and the image is SYS\$SYSTEM:LOGINOUT.EXE. If you specify this option in your call to \$CREPRC, the process created by the call is logged in under the username associated with the creating process. If this option is not specified for an interactive process, SYS\$SYSTEM:LOGINOUT.EXE prompts the user for the username and password to be associated with the process. The prompts are displayed at the SYS\$INPUT device.

Note that options PRC\$_BATCH, PRC\$_INTER, PRC\$_UAF, PRC\$_NETWRK and PRC\$_NOPASSWORD are intended for use by DIGITAL software. Complete documentation of the possible ramifications of their use will not be provided.

DESCRIPTION

The calling process must have

- DETACH privilege to create any of the following types of process:
 - A detached process with a UIC that is different from the UIC of the calling process
 - A batch process
 - A network process
- ALTPRI privilege to create a subprocess with a higher base priority than the calling process
- SETPRV privilege to create a process with privileges that the calling process does not have
- PSWAPM privilege to create a process with process swap mode disabled
- NOACNT privilege to create a process with accounting functions disabled
- NETMBX privilege to create a network connect object

A detached process is a fully independent process. For example, the process that the system creates when a user logs in is a detached process.

System Service Descriptions

\$CREPRC

A subprocess, on the other hand, is related to its creator in a treelike structure; it receives a portion of the creating process's resource quotas and must terminate before the creating process. The `uic` argument or the `PRC$_DETACH` flag controls whether the created process is a subprocess or a detached process.

The \$CREPRC service requires system dynamic memory.

The number of subprocesses that a process can create is controlled by the subprocess (PRCLM) quota; this quota is returned when a subprocess is deleted.

The number of detached processes that a process can create with the same username is controlled by the MAXDETACH entry in the user authorization file (UAF).

When a subprocess is created, the value of any deductible quota is subtracted from the total value the creator has available; and when the subprocess is deleted, the unused portion of any deductible quota is added back to the total available to the creator. Any pooled quota value is shared by the creator and all its subprocesses.

Some error conditions are not detected until the created process executes. These conditions include an invalid or nonexistent image; invalid `SYS$INPUT`, `SYS$OUTPUT`, or `SYS$ERROR` logical name equivalence; inadequate quotas; or insufficient privilege to execute the requested image.

All subprocesses created by a process must terminate before the creating process can be deleted. If subprocesses exist when their creator is deleted, they are automatically deleted.

A created process will be unable to run an image that calls the Run-Time Library procedure `LIB$DO_COMMAND` unless the process was created with the `image` argument specifying `SYS$SYSTEM:LOGINOUT.EXE`. This is true because `SYS$SYSTEM:LOGINOUT.EXE` causes a command language interpreter to be mapped into the created process, a prerequisite for calling `LIB$DO_COMMAND`.

A detached process is considered an interactive process only if (1) the process is created with the `PRC$_INTER` option specified and (2) `SYS$INPUT` is not defined as a file-oriented device.

CONDITION VALUES RETURNED

`SS$_NORMAL`

Service successfully completed.

`SS$_ACCVIO`

The caller cannot read a specified input string or string descriptor, the privilege list, or the quota list; or the caller cannot write the process identification.

`SS$_DUPLNAM`

The specified process name duplicates one already specified within that group.

System Service Descriptions

SCREPRC

SS\$_EXQUOTA

At least one of the three following conditions is true:

- The process has exceeded its quota for the creation of subprocesses
- A quota value specified for the creation of a subprocess exceeds the creator's corresponding quota
- The quota is deductible and the remaining quota for the creator would be less than the minimum

SS\$_INSFMEM

Insufficient system dynamic memory is available for the requested operation.

SS\$_IVLOGNAM

At least one of the following two conditions is true:

- The specified process name has a length of 0 or has more than 15 characters
- Or the specified image name, input name, output name, or error name has more than 255 characters.

SS\$_IVQUOTAL

The quota list is not in the proper format.

SS\$_IVSTSFLG

A reserved status flag was set.

SS\$_NOPRIV

The caller has violated one of the privilege restrictions.

SS\$_NOSLOT

No process control block is available; in other words, the maximum number of processes that can exist concurrently in the system has been reached.

SS\$_INSSWAPSPACE

Insufficient swap space exists to create the process.

SS\$_EXPRCLM

The creation of a detached process failed because the creating process has already reached its limit for the creation of detached processes. This limit is established by the MAXDETACH quota in the creating process's user authorization file (UAF).

\$CREATE_RDB—Create Rights Database

The Create Rights Database service initializes a rights database.

FORMAT **SY\$CREATE_RDB** [*sysid*]

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS ***sysid***

VMS Usage: **system_access_id**
type: **quadword (unsigned)**
access: **read only**
mechanism: **by reference**

System identification value associated with the rights database when \$CREATE_RDB completes execution. The ***sysid*** argument is the address of quadword containing the system identification value. If ***sysid*** is omitted, the current system time in 64-bit format is used.

DESCRIPTION

The Create Rights Database service initializes a rights database. The database name is the file equated to the logical name RIGHTSLIST, which must be defined as a system logical name at executive mode. If the logical name does not exist, the database is named SYS\$SYSTEM:RIGHTSLIST.DAT.

If the database already exists, \$CREATE_RDB fails with the error RMS\$_FEX.

Write access to the rights database is required to use this service. If the database is in SYS\$SYSTEM (which is the default) SYSPRV privilege is needed to grant write access to the database.

System Service Descriptions

\$CREATE_RDB

CONDITION VALUES RETURNED

SS\$_NORMAL

Service is successfully completed.

SS\$_ACCVIO

The sysid argument cannot be read by the caller.

SS\$_INSFMEM

Insufficient process dynamic memory is available to open the rights database.

RMS\$_FEX

A rights database already exists. To create a new one, you must explicitly delete or rename the old one.

RMS\$_PRV

The user does not have write access to SYSS\$SYSTEM:.

Since the rights database is an indexed file accessed with VAX RMS, this service may also return RMS status codes associated with operations on indexed files. Please refer to the *VAX Record Management Services Reference Manual* for descriptions of these status codes.

\$CRETVA—Create Virtual Address Space

The Create Virtual Address Space service adds a range of demand-zero allocation pages to a process's virtual address space for the execution of the current image.

FORMAT **SY\$CRETVA** *inadr*, [*retadr*], [*acmode*]

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS *inadr*

VMS Usage: **address_range**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Address of a two-longword array containing the starting and ending virtual addresses of the pages to be created. If the starting and ending virtual addresses are the same, a single page is created. Only the virtual page number portion of the virtual addresses is used; the low-order 9 bits are ignored.

retadr

VMS Usage: **address_range**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference—array reference or descriptor**

Address of a two-longword array to receive the starting and ending virtual addresses of the pages actually created.

acmode

VMS Usage: **access_mode**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Access mode and protection for the new pages. The **acmode** argument is a longword containing the access mode. The \$PSLDEF macro defines the following symbols for the four access modes:

System Service Descriptions

\$CRETVA

Symbol	Access mode
PSL\$C_KERNEL	Kernel mode
PSL\$C_EXEC	Executive mode
PSL\$C_SUPER	Supervisor mode
PSL\$C_USER	User mode

The most privileged access mode used is the access mode of the caller. The protection of the pages is read/write for the resultant access mode and those more privileged.

DESCRIPTION

The process's paging file quota (PGFLQUOTA) must be sufficient to accommodate the increased size of the virtual address space.

Pages are created starting at the address contained in the first longword of the location addressed by the **inadr** argument and ending with the second longword. The ending address can be lower than the starting address. The **retadr** argument indicates the byte addresses of the pages created.

If an error occurs while creating pages, the **retadr** argument, if specified, indicates the pages that were successfully created before the error occurred. If no pages were created, both longwords of the **retadr** argument contain a -1.

If \$CRETVA creates pages that already exist, the service deletes those pages if they are not owned by a more privileged access mode than that of the caller. Any such deleted pages are reinitialized as demand-zero pages.

Note that the Expand Program/Control Region (\$EXPREG) service also adds pages to a process's virtual address space.

Note: Do not use the \$CNTREG, or \$CRETVA system services in conjunction with other user-written procedures and/or DIGITAL-supplied procedures (including run-time library procedures). These system services provide no means to communicate a change in virtual address space with other routines. DIGITAL recommends that you use either \$EXPREG or the Run-time Library Procedure Allocate Virtual Memory (LIB\$GET_VM) to get memory. You can find documentation on LIB\$GET_VM in the *VAX/VMS Run-Time Library Routines Reference Manual*. When using \$DELTVA, you should take care to only delete pages that you have specifically created.

System Service Descriptions

SECRETVA

CONDITION VALUES RETURNED

SS\$_NORMAL

Service successfully completed.

SS\$_ACCVIO

The **inadr** argument cannot be read by the caller, or the **retadr** argument cannot be written by the caller.

SS\$_EXQUOTA

The process has exceeded its paging file quota.

SS\$_INSFWSL

The process's working set limit is not large enough to accommodate the increased size of the virtual address space.

SS\$_NOPRIV

A page in the specified range is in the system address space.

SS\$_PAGOWNVIO

A page in the specified range already exists and can not be deleted because it is owned by a more privileged access mode than that of the caller.

SS\$_VASFULL

The process's virtual address space is full; no space is available in the page tables for the requested pages.

\$CRMPSC—Create and Map Section

The Create and Map Section service allows a process to associate (map) a section of its address space with (1) a specified section of a file (a disk file section) or (2) with specified physical addresses represented by page frame numbers (a page frame section).

This service also allows the process to create either type of section, and to specify that that section be available only to the creating process (private section) or to all processes that map to it (global section).

FORMAT	SY\$CRMPSC <i>[inadr] , [retadr] , [acmode] , [flags] , [gsdnam] , [ident] , [relpag] , [chan] , [pagcnt] , [vbn] , [prot] , [pfc]</i>
---------------	---

RETURNS	VMS Usage: cond_value type: longword (unsigned) access: write only mechanism: by value
----------------	---

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS	<i>inadr</i> VMS Usage: address_range type: longword (unsigned) access: read only mechanism: by reference Starting and ending virtual addresses into which the section is to be mapped. The <i>inadr</i> is the address of a two-longword array containing, in order, the starting and the ending process virtual addresses. Only the virtual page number portion of each virtual address is used; the low-order 9 bits are ignored.
------------------	---

If the starting and ending virtual addresses are the same, a single page is mapped, unless the SEC\$M_EXPREG bit in the ***flags*** argument is set. If this bit is set, the specified address simply determines whether the section is mapped in the program (P0) or control (P1) region.

If ***inadr*** is not specified or is specified as 0, the section is not mapped.

retadr
VMS Usage: **address_range**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference—array reference or descriptor**
Starting and ending process virtual addresses into which the section was actually mapped by \$CRMPSC. The ***retadr*** is the address of a two-longword array containing, in order, the starting and ending process virtual addresses.

System Service Descriptions

SCRMPS

acmode

VMS Usage: **access_mode**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Access mode that is to be the owner of the pages created during the mapping. The **acmode** argument is a longword containing the access mode. The \$PSLDEF macro defines the following symbols for the four access modes:

Symbol	Access mode
PSL\$_KERNEL	Kernel mode
PSL\$_EXEC	Executive mode
PSL\$_SUPER	Supervisor mode
PSL\$_USER	User mode

The most privileged access mode used is the access mode of the caller.

flags

VMS Usage: **mask_longword**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Flag mask specifying the type of section to be created or mapped to, as well as its characteristics. The **flags** argument is a longword bit vector wherein each bit corresponds to a flag. The \$SECDEF macro defines a symbolic name for each flag. The **flags** argument is constructed by performing a logical OR operation on the symbol names for all desired flags. The following lists each symbol name, its description, and the default value that it supersedes.

Flag	Description
SEC\$_GBL	Pages form a global section. The default is private section.
SEC\$_CRF	Pages are copy-on-reference. By default, pages are shared.
SEC\$_DZRO	Pages are demand-zero pages. By default, they are not zeroed when copied.
SEC\$_EXPREG	Pages are mapped into the first available space. By default, pages are mapped into the range specified by the inadr argument.
SEC\$_WRT	Pages form a read/write section. By default, pages form a read-only section.
SEC\$_PERM	Pages are permanent. By default, pages are temporary.
SEC\$_PFNMAP	Pages form a page-frame section. By default, pages form a disk-file section. Pages mapped by using SEC\$_PFNMAP are not included in or charged against the process's working set; they are always valid. Do not lock these pages in the working set by using \$LKWSET; this may result in a machine check.
SEC\$_SYSGBL	Pages form a system global section. By default, pages form a group global section.

System Service Descriptions

\$CRMPSC

Flag	Description
SEC\$_PAGFIL	Pages form a global page-file section. By default, pages form a disk-file section.
SEC\$_EXECUTE	Pages are mapped if the caller has execute access. This flag is valid only (1) when specified from executive or kernel mode access and (2) when the SEC\$_GBL flag is also specified. By default, the pages are mapped regardless of whether the caller has execute access.

gsdnam

VMS Usage: **section_name**

type: **character-coded text string**

access: **read only**

mechanism: **by descriptor—fixed length string descriptor**

Name of the global section. The **gsdnam** is the address of a character string descriptor pointing to this name string. Section 11.6.6.1 describes the format of this name string.

For group global sections, VAX/VMS interprets the UIC group as part of the global section name; thus, the names of global sections are unique to UIC groups.

ident

VMS Usage: **section_id**

type: **quadword (unsigned)**

access: **read only**

mechanism: **by reference**

Identification value specifying the version number of a global section, and, for processes mapping to an existing global section, the criteria for matching the identification. The **ident** argument is the address of a quadword structure containing three fields.

The version number is in the second longword. The version number contains two fields: a minor identification in the low-order 24 bits and a major identification in the high-order 8 bits. Values for these fields can be assigned by installation convention to differentiate versions of global sections. If no version number is specified when a section is created, processes that specify a version number when mapping cannot access the global section.

The first longword specifies, in its low-order 3 bits, the matching criteria. The valid values, symbolic names by which they can be specified, and their meanings are:

Value/Name	Match Criteria
0 SEC\$_MATALL	Match all versions of the section
1 SEC\$_MATEQU	Match only if major and minor identifications match
2 SEC\$_MATLEQ	Match if the major identifications are equal and the minor identification of the mapper is less than or equal to the minor identification of the global section

The match control field is ignored when a section is mapped at creation time.

If **ident** is not specified or is specified as 0 (the default), the version number and match control fields default to 0.

System Service Descriptions

\$CRMPSC

relpag

VMS Usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Relative page number within the global section of the first page in the section to be mapped. The **relpag** argument is a longword containing this page number.

This argument is used only for global sections. If **relpag** is not specified or is specified as 0 (the default), the global section is mapped beginning with the first virtual block in the file. This argument must be 0 for demand-zero sections in memory shared by multiple processors.

chan

VMS Usage: **channel**
type: **word (unsigned)**
access: **read only**
mechanism: **by value**

Number of the channel on which the file has been accessed. The **chan** argument is a word containing this number.

The file must have been accessed with a VAX RMS \$OPEN macro; the file options parameter (FOP) in the FAB must indicate a user file open (UFO keyword). The access mode at which the channel was opened must be the same or less privileged than the access mode of the caller.

pagcnt

VMS Usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Number of pages in the section. The **pagcnt** argument is a longword containing this number.

The specified page count is compared with the number of pages in the section file; if they are different, the lower value is used. If the page count is not specified or is specified as 0 (the default), the size of the section file is used. However, for physical page frame sections, this argument must not be 0.

vbn

VMS Usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Virtual block number in the file that marks the beginning of the section. The **vbn** argument is a longword containing this number. If **vbn** is not specified or is specified as 0 (the default), the section is created beginning with the first virtual block in the file.

If you specified page frame number mapping (by setting the SEC\$M_PFNMAP flag), the **vbn** argument specifies the page frame number where the section begins in memory.

System Service Descriptions

\$CRMPSC

Table SYS-1 depicts which arguments are required and which are optional for three different uses of the \$CRMPSC service.

Table SYS-1 Required and Optional Arguments for the \$CRMPSC Service

Argument	Create/Map Global Section	Map Global ¹ Section	Create/Map Private Section
inadr	Optional ²	Required	Required
retadr	Optional	Optional	Optional
acmode	Optional	Optional	Optional
flags			
SEC\$_M_GBL	Required	Ignored	—
SEC\$_M_CRF ³	Optional	Not used	Optional
SEC\$_M_DZRO ³	Optional	Not used	Optional
SEC\$_M_EXPREG	Optional	Optional	Optional
SEC\$_M_PERM	Optional ²	Not used	Not used
SEC\$_M_PFNMAP	Optional	Not used	Not used
SEC\$_M_SYSGBL	Optional	Optional	Not used
SEC\$_M_WRT	Optional	Optional	Optional
SEC\$_M_PAGFIL	Optional	Not used	Not used
gsdnam	Required	Required	Not used
ident	Optional	Optional	Not used

¹The Map Global Section (\$MGBLSC) service maps an existing global section.

²The **inadr** argument can be omitted only if you wish to create but not map a global section; however, in such a case you must make the section permanent because temporary sections are automatically deleted when no processes are mapped to them. The **inadr** argument cannot be omitted for demand-zero sections in memory shared by multiple processors.

³For physical page frame sections: **vbn** specifies the starting page frame number; **chan** must be zero; **relpag** and **pfc** are not used; and the SEC\$_M_CRF and SEC\$_M_DZRO flag bit settings are invalid. For page-file sections, **chan** must be zero, and **relpag** and **pfc** are not used.

System Service Descriptions

\$CRMPSC

Table SYS-1 (Cont.) Required and Optional Arguments for the \$CRMPSC Service

Argument	Create/Map Global Section	Map Global ¹ Section	Create/Map Private Section
relpag ³	Optional	Optional	Not used
chan ³	Required		Required
pagcnt	Required		Required
vbn ³	Optional		Optional
prot	Optional		Not used
pfc ³	Optional ⁴		Optional

¹The Map Global Section (\$MGBLSC) service maps an existing global section.

³For physical page frame sections: **vbn** specifies the starting page frame number; **chan** must be zero; **relpag** and **pfc** are not used; and the SEC\$M_CRF and SEC\$M_DZRO flag bit settings are invalid. For page-file sections, **chan** must be zero, and **relpag** and **pfc** are not used.

⁴This argument is not used for global sections in memory shared by multiple processors.

prot

VMS Usage: **file_protection**

type: **longword (unsigned)**

access: **read only**

mechanism: **by value**

Numeric value representing the protection mask to be applied to the global section. This value is ORed with the protection mask associated with the file; if the file protection does not allow access to a particular category of user and the protection mask allows access, access is denied.

The mask contains four 4-bit fields. Bits are read from right to left in each field. The following diagram depicts the mask.

WORLD				GROUP				OWNER				SYSTEM			
D	E	W	R	D	E	W	R	D	E	W	R	D	E	W	R
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

ZK-1706-84

Cleared bits indicate that read, write, execute, and delete access, in that order, are granted to the particular category of user.

Only read, write, and execute access are meaningful for section protection. Delete access bits are ignored. \$CRMPSC checks the execute access bit only for calls from executive or kernel mode.

System Service Descriptions

\$CRMPSC

If the **prot** argument is not specified or is specified as 0, read access and write access are granted to all users.

pfc

VMS Usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Page fault cluster size. If specified, the cluster size indicates how many pages are to be brought into memory when a page fault occurs for a single page. This argument is not used for physical page frame sections or for global sections in memory shared by multiple processors.

DESCRIPTION

If the section pages are copy-on-reference, the process must have sufficient paging file quota (PGFLQUOTA). The system-wide number of global page-file pages is limited by the SYSGEN parameter GBLPAGFIL.

Note that the PFNMAP privilege is not required to map to an existing page frame section, and the SHMEM privilege is not required to map to an existing global section in memory shared by multiple processors.

Creating a disk file section involves defining all or part of a disk file as a section. Mapping a disk file section involves making a correspondence between virtual blocks in the file and pages in the caller's virtual address space. If the \$CRMPSC service specifies a global section that already exists, the service maps it.

If \$CRMPSC specifies a global section and the SS\$NOPRIV condition value is returned, the process may not have the required privilege to create that section. In order to create global sections the process must have the following privileges:

- The SYSGBL privilege to create a system global section.
- The PRMGBL privilege to create a permanent global section.
- The PFNMAP privilege to create a page frame section.
- The SHMEM privilege to create a global section in memory shared by multiple processors.

Note that the PFNMAP privilege is not required to map an existing page frame section, and the SHMEM privilege is not required to map an existing global section in memory shared by multiple processors.

Depending on the actual operation requested, certain arguments are required or optional. Table SYS-1 summarizes how the \$CRMPSC service interprets the arguments passed to it, and under what circumstances it requires or ignores arguments.

When \$CRMPSC maps a section, it calls the Create Virtual Address Space (\$CRETVA) service to add the pages specified by the **inadr** argument or requested by the SEC\$M_EXPREG flag bit setting to the process's virtual address space. The virtual addresses can be in the program (P0) region or the control (P1) region.

The \$CRMPSC service returns the virtual addresses of the pages created in the **retadr** argument, if specified. The section is mapped from a low address to a high address, regardless of whether the section is mapped in the program or control region.

System Service Descriptions

\$CRMPSC

If an error occurs during the mapping of a global section, the **retadr** argument, if specified, indicates the pages that were successfully mapped when the error occurred. If no pages were mapped, both longwords of the **retadr** argument contain -1.

If the global section is permanent, it is not deleted if the mapping operation fails.

The **SEC\$_PFNMAP** flag setting identifies the memory for the section as starting at the page frame number specified in the **vbn** argument and extending for the number of pages specified in the **pagcnt** argument. Setting the **SEC\$_PFNMAP** flag places restrictions on the following arguments:

relpag	Does not apply
chan	Must be zero
pagcnt	Must be specified; cannot be zero
vbn	Specifies first page frame to be mapped
pfc	Does not apply

Setting the **SEC\$_PFNMAP** flag also places restrictions on these other flag values:

SEC\$_CRF	Must be 0
SEC\$_DZRO	Must be 0
SEC\$_PERM	Must be 1 if the flags SEC\$_GBL or SEC\$_SYSGBL are set

Setting the **SEC\$_PAGFIL** flag places the following restrictions on the following flags:

SEC\$_CRF	Must be 0
SEC\$_GBL	Must be 1
SEC\$_PFNMAP	Must be 0

The **flag** argument bits 4 through 13 (inclusive) and 18 through 31 (inclusive) must be 0.

The flag bit **SEC\$_WRT** applies only to the way in which the newly created section is mapped. For a file to be made writeable, the channel used to open the file must allow write access to the file.

If the flag bit **SEC\$_SYSGBL** is set, the flag bit **SEC\$_GBL** must be set also.

The creation of a global section in memory shared by more than one processor requires GLOBAL SECTION PORT quota. This quota is acquired using the **SYSGEN** command **SHARE**; it is returned when the global section is deleted or when the **SYSGEN** command **SHARE** is reissued after the processor on that port is rebooted.

A global section in memory shared by more than one processor can only be deleted by the creator processor. If a global section is marked as having no creator processor, you must perform the following two actions before attempting to delete the global section: (1) reboot all processors and (2) reinitialize the MA780 shared memory.

System Service Descriptions

\$CRMPSC

A global section is marked as having no creator processor when you rebootstrap a processor and reconnect it to an MA780 shared memory without reinitializing the MA780. In such a circumstance, the SYSGEN utility does cleanup for the processor, and this cleanup causes all global sections created by processes running on the processor to be marked as having no creator processor; the data structures that allow the data in the global section pages to be written back into the disk file no longer exist.

CONDITION VALUES RETURNED

SS\$_NORMAL	Service successfully completed. The specified global section already existed and has been mapped.
SS\$_CREATED	Service successfully completed. The specified global section did not previously exist and has been created.
SS\$_ACCVIO	The inadr argument, gsdnam argument, or name descriptor cannot be read by the caller; or the retadr argument cannot be written by the caller.
SS\$_ENDOFFILE	Warning. The starting virtual block number specified is beyond the logical end-of-file, or the value in the relpag argument is greater than or equal to the value in the pagcnt argument.
SS\$_EXBYTLM	The process has exceeded the byte count quota; the system was unable to map the requested file.
SS\$_EXGBLPAGFIL	The process has exceeded the system-wide limit on global page-file pages; no part of the section was mapped.
SS\$_EXPORTQUOTA	The process has exceeded the number of global sections that processes on this port of the multiport (shared) memory can create.
SS\$_EXQUOTA	The process exceeded its paging file quota while creating copy-on-reference or page-file-backing-store pages.
SS\$_GPTFULL	There is no more room in the system global page table to set up page table entries for the section.
SS\$_GSDFULL	There is no more room in the system space allocated to maintain control information for global sections.
SS\$_ILLPAGCNT	The page count value is negative or is zero for a physical page frame section.
SS\$_INSMEM	Not enough pages are available in the specified shared memory to create the section.
SS\$_INSFWSL	The process's working set limit is not large enough to accommodate the increased size of the address space.
SS\$_INTERLOCK	The bit map lock for allocating global sections from the specified shared memory is locked by another process.
SS\$_IVCHAN	An invalid channel number was specified, that is, a channel number of 0 or a number larger than the number of channels available.

System Service Descriptions

\$CRMPSC

SS\$_IVCHNLSEC	The channel number specified is currently active.
SS\$_IVLOGNAM	The specified global section name has a length of 0 or has more than 15 characters.
SS\$_IVLVEC	The specified section was not installed using the /PROTECT qualifier.
SS\$_IVSECFLG	An invalid flag, a reserved flag, a flag requiring a privilege you lack, or an invalid combination of flags was specified.
SS\$_IVSECIDCTL	The match control field of the global section identification is invalid.
SS\$_NOTFILEDEV	The device is not a file-oriented, random-access, or directory device.
SS\$_NOPRIV	<p>The process does not have the privileges to create a system global section (SYSGBL) or a permanent group global section (PRMGBL).</p> <p>The process does not have the privilege to create a section starting at a specific physical page frame number (PFNMAP).</p> <p>The process does not have the privilege to create a global section in memory shared by multiple processors (SHMEM).</p> <p>A page in the input address range is in the system address space.</p> <p>The specified channel is not assigned or was assigned from a more privileged access mode.</p>
SS\$_NOSHMBLOCK	No shared memory control block for global sections is available.
SS\$_NOWRT	You cannot write to the section because the flag bit SEC\$_WRT is set, the file is read only, and the flag bit SEC\$_CRF is not set.
SS\$_PAGOWNVIO	A page in the specified input address range is owned by a more privileged access mode.
SS\$_SECTBLFUL	There are no entries available in the system global section table.
SS\$_SHMNOTCNCT	The shared memory named in the gsdnam argument is not known to the system. This error can be caused by a spelling error in the string, an improperly assigned logical name, or the failure to identify the memory as shared at system generation time.
SS\$_TOOMANYLNAM	Logical name translation of the gsdnam argument exceeded the allowed depth.
SS\$_VASFULL	The process's virtual address space is full; no space is available in the page tables for the pages created to contain the mapped global section.

System Service Descriptions

\$DACEFC

\$DACEFC—Disassociate Common Event Flag Cluster

The Disassociate Common Event Flag Cluster service releases the calling process's association with a common event flag cluster.

FORMAT **SY\$DACEFC** *efn*

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENT *efn*

VMS Usage: **ef_number**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Number of any event flag in the common cluster to be disassociated. The *efn* argument is a longword containing this number. The number must be in the range of 64 through 95 for cluster 2, and 96 through 127 for cluster 3.

DESCRIPTION

The count of processes associated with the cluster is decreased for each process that disassociates. When the image that associated with a cluster exits, the system disassociates the cluster. The cluster is automatically deleted when the count of processes associated with a temporary cluster or with a permanent cluster that is marked for deletion reaches zero.

If a process issues this service specifying an event flag cluster with which it is not associated, the service completes successfully.

**CONDITION
VALUES
RETURNED**

SS\$_NORMAL

Service successfully completed.

SS\$_ILLEFC

An illegal event flag number was specified. The number must be in the range of event flags 64 through 127.

SS\$_INTERLOCK

The bit map lock for allocating common event flag clusters from the specified shared memory is locked by another process.

\$DALLOC—Deallocate Device

The Deallocate Device service deallocates a previously allocated device. The issuing process relinquishes exclusive use of the device thus allowing other processes to assign or allocate that device.

FORMAT **SYSDALLOC** [*devnam*],[*acmode*]

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS *devnam*

VMS Usage: **device_name**
type: **character-coded text string**
access: **read only**
mechanism: **by descriptor—fixed length string descriptor**

Name of the device to be deallocated. The *devnam* argument is the address of a character string descriptor pointing to the device name string. The string may be either a physical device name or a logical name. If it is a logical name, it must translate to a physical device name.

If no device name is specified, all devices allocated by the process from access modes equal to or less privileged than that specified are deallocated.

acmode

VMS Usage: **access_mode**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Access mode on behalf of which the deallocation is to be performed. The *acmode* argument is a longword containing the access mode. The \$PSLDEF macro defines the following symbols for the four access modes.

Symbol	Access mode
PSL\$C_KERNEL	Kernel mode
PSL\$C_EXEC	Executive mode
PSL\$C_SUPER	Supervisor mode
PSL\$C_USER	User mode

The most privileged access mode used is the access mode of the caller.

System Service Descriptions

\$DALLOC

DESCRIPTION

An allocated device can be deallocated only from access modes equal to or more privileged than the access mode from which the original allocation was made.

This service will not deallocate a device if, at the time of deallocation, the issuing process has one or more I/O channels assigned to the device; in such a case, the device remains allocated.

At image exit, the system automatically deallocates all devices that were allocated at user mode.

If an attempt is made to deallocate a mailbox, success is returned but no operation is performed.

CONDITION VALUES RETURNED

SS\$_NORMAL	Service successfully completed.
SS\$_ACCVIO	The device name string or string descriptor cannot be read by the caller.
SS\$_DEVASSIGN	Warning. The device cannot be deallocated because the process still has channels assigned to it.
SS\$_DEVNOTALLOC	Warning. The device is not allocated to the requesting process.
SS\$_IVDEVNAM	No device name string was specified, or the device name string contains invalid characters.
SS\$_IVLOGNAM	The device name string has a length of 0 or has more than 63 characters.
SS\$_NONLOCAL	Warning. The device is on a remote node.
SS\$_NOPRIV	The device was allocated from a more privileged access mode.
SS\$_NOSUCHDEV	Warning. The specified device does not exist in the host system.

\$DASSGN—Deassign I/O Channel

The Deassign I/O Channel service deassigns (releases) an I/O channel that was acquired using the Assign I/O Channel (\$ASSIGN) service.

FORMAT **SY\$DASSGN** *chan*

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENT *chan*

VMS Usage: **channel**
type: **word (unsigned)**
access: **read only**
mechanism: **by value**

Number of the I/O channel to be deassigned. The *chan* argument is a word containing this number.

DESCRIPTION

An I/O channel can be deassigned only from an access mode equal to or more privileged than the access mode from which the original channel assignment was made.

When a channel is deassigned, any outstanding I/O requests on the channel are canceled. If a file is open on the specified channel, the file is closed.

If a mailbox was associated with the device when the channel was assigned, the link to the mailbox is cleared.

If the I/O channel was assigned for a network operation, the network link is disconnected.

If the specified channel is the last channel assigned to a device that has been marked for dismounting, the device is dismounted.

I/O channels assigned from user mode are automatically deassigned at image exit.

System Service Descriptions

\$DASSGN

CONDITION VALUES RETURNED

SS\$_NORMAL

Service successfully completed.

SS\$_IVCHAN

An invalid channel number was specified, that is, a channel number of 0 or a number larger than the number of channels available.

SS\$_NOPRIV

The specified channel is not assigned or was assigned from a more privileged access mode.

\$DCLAST—Declare AST

The Declare AST service queues an AST for the calling access mode or for a less privileged access mode. For example, a routine executing in supervisor mode can declare an AST for either supervisor or user mode.

FORMAT **SY\$DCLAST** *astadr* [,*astprm*] [,*acmode*]

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS ***astadr***

VMS Usage: **ast_procedure**
type: **procedure entry mask**
access: **call without stack unwinding**
mechanism: **by reference**

AST service routine to be executed. The *astadr* argument is the address of the entry mask of this routine.

astprm

VMS Usage: **user_arg**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

AST parameter to be passed to the AST routine specified by the *astadr* argument. The *astprm* argument is a longword containing this parameter.

acmode

VMS Usage: **access_mode**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Access mode for which the AST is to be declared. The most privileged access mode used is the access mode of the caller. The resultant mode is the access mode for which the AST is declared.

System Service Descriptions

\$DCLAST

DESCRIPTION The \$DCLAST service requires system dynamic memory and uses the process's AST limit (ASTLM) quota.

The service does not validate the address of the AST service routine. If an illegal address (such as 0) is specified, an access violation occurs when the AST service routine is given control.

**CONDITION
VALUES
RETURNED**

SS\$_NORMAL
SS\$_EXQUOTA

Service successfully completed.

The process has exceeded its AST limit (ASTLM) quota.

SS\$_INSFMEM

Insufficient system dynamic memory is available to complete the service.

\$DCLCMH—Declare Change Mode or Compatibility Mode Handler

The Declare Change Mode or Compatibility Mode Handler service specifies the address of a routine to receive control when (1) a Change Mode to User or Change Mode to Supervisor instruction trap occurs, or (2) a compatibility mode fault occurs.

FORMAT **SYSDCLCMH** *adres* , [*prvhnd*] , [*type*]

RETURNS

VMS Usage: **cond_value**
 type: **longword (unsigned)**
 access: **write only**
 mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS

adres

VMS Usage: **procedure**
 type: **procedure entry mask**
 access: **call without stack unwinding**
 mechanism: **by reference**

Routine to receive control when a change mode trap or a compatibility mode fault occurs. The **adres** argument is the address of the entry mask to this routine.

If **adres** specifies 0, \$DCLCMH clears a previously declared handler.

prvhnd

VMS Usage: **address**
 type: **longword (unsigned)**
 access: **write only**
 mechanism: **by reference**

Address of a previously declared handler. The **prvhnd** argument is the address of a longword containing the address of the previously declared handler.

type

VMS Usage: **longword_unsigned**
 type: **longword (unsigned)**
 access: **read only**
 mechanism: **by value**

Handler type indicator. The **type** argument is a longword value. The value 0 (the default) indicates that a change mode handler is to be declared for the access mode at which the request is issued; the value 1 specifies that a compatibility mode handler is to be declared.

System Service Descriptions

\$DCLCMH

DESCRIPTION

A change mode handler provides users with a dispatching mechanism similar to that used for system service calls. It allows a routine that executes in supervisor mode to be called from user mode. The change mode handler is declared from supervisor mode; then when the process executing in user mode issues a Change Mode to Supervisor instruction, the change mode handler receives control and executes in supervisor mode.

Compatibility mode handlers are used by the operating system to bypass normal condition handling procedures when an image executing in compatibility mode causes a compatibility mode exception.

Before the change mode handler exits, it must push the saved PC and PSL onto the stack from the location CTL\$AL_CMCNTX, and it must exit by issuing an REI instruction.

A change mode handler can be declared only from user or supervisor mode.

CONDITION VALUES RETURNED

SS\$_NORMAL

Service successfully completed.

SS\$_ACCVIO

The longword to receive the address of the previous change mode handler cannot be written by the caller.

\$DCLEXH—Declare Exit Handler

The Declare Exit Handler service declares an exit handling routine that will receive control when an image exits. Image exit normally occurs when the image currently executing in a process returns control to the operating system. Image exit may also occur when the Exit (\$EXIT) or Force Exit (\$FORCEX) services are called.

FORMAT **SYS\$DCLEXH** *desblk*

RETURNS

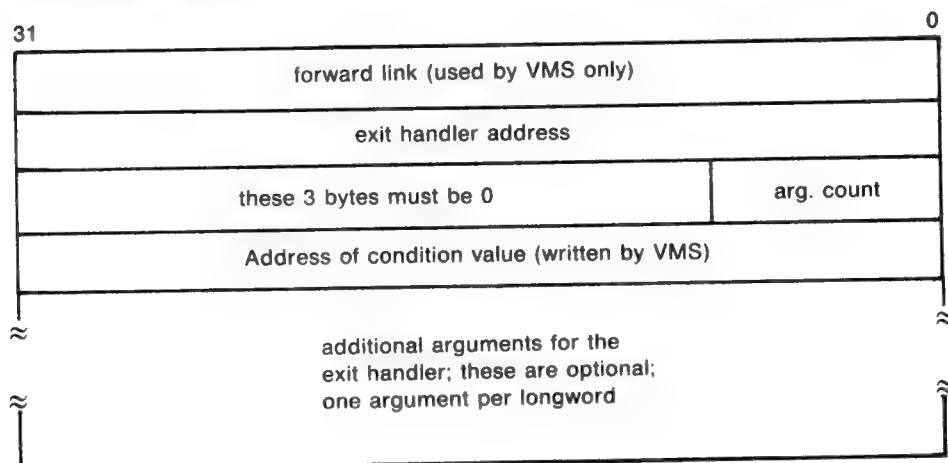
VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENT *desblk*

VMS Usage: **exit_handler_block**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Exit handler control block. The **desblk** is the address of this control block. This control block, which describes the exit handler, is depicted in the following diagram:



ZK-1714-84

System Service Descriptions

\$DCLEXH

DESCRIPTION

Exit handlers are described by exit control blocks. The operating system maintains a separate list of these control blocks for user, supervisor, and executive modes. The \$DCLEXH service adds the description of an exit handler to the front of one of these lists. The actual list to which the exit control block is added is determined by the access mode of the caller.

This service can only be called from user, supervisor, and executive modes.

At image exit, the exit handlers declared from user mode are called first; they are called in the reverse order from which they were declared.

Each exit handler is executed only once; it must be redeclared before it can be executed again. The exit handling routine is called as a normal procedure with the argument list specified in the 3rd through nth longwords of the exit control block. The first argument is the address of a longword to receive a system status code indicating the reason for exit; the system always fills in this longword before calling the exit handler.

The Cancel Exit Handler (\$CANEXH) service removes an exit control block from the list.

CONDITION VALUES RETURNED

SS\$_NORMAL

Service successfully completed.

SS\$_ACCVIO

The first longword of the exit control block cannot be written by the caller.

SS\$_IVSSRQ

The call to the service is invalid because it was made from kernel mode.

SS\$_NOHANDLER

Warning. The exit handler control block address was not specified or was specified as 0.

\$DELLNM—Delete Logical Name

The Delete Logical Name service deletes all logical names with the specified name at the specified access mode or outer access mode, or it deletes all the logical names with the specified access mode or outer access mode in a specified table. If any logical names being deleted are also the names of logical name tables, then all of the logical names contained within those tables and all of their subtables are also deleted.

FORMAT **SYSS\$DELLNM** *tabnam* ,[*lognam*] ,[*acmode*]

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS *tabnam*

VMS Usage: **logical_name**
type: **character-coded text string**
access: **read only**
mechanism: **by descriptor—fixed length string descriptor**

Name of a logical name table or name of a list of tables in which to search for the logical name that is to be deleted. The **tabnam** argument is the address of a descriptor that points to the table name. This argument is required.

If **tabnam** is not the name of a logical name table, it is assumed to be a logical name and is translated iteratively until either the name of a logical name table is found or the number of translations allowed by the system has been performed.

If **tabnam** translates to the name of a list of tables, \$DELLNM does the following:

- If the **lognam** argument is specified, \$DELLNM searches (in order) each table in the list until it finds the first table that contains the specified logical name. If the logical name is at the specified access mode, \$DELLNM then deletes occurrences of the logical name at the specified access mode and at outer access modes within the table.
- If the **lognam** argument is not specified, \$DELLNM deletes all of the logical names at the specified access mode or at outer access modes from the first table in the list whose access mode is equal to or less privileged than the caller's access mode.

System Service Descriptions

\$DELLNM

lognam

VMS Usage: **logical_name**

type: **character-coded text string**

access: **read only**

mechanism: **by descriptor—fixed length string descriptor**

Logical name to be deleted. The **lognam** argument is the address of a descriptor that points to the logical name string.

acmode

VMS Usage: **access_mode**

type: **byte (unsigned)**

access: **read only**

mechanism: **by reference**

Access mode to be used in the delete operation. The **acmode** argument is the address of a byte containing this access mode. The **\$PSLDEF** macro defines symbolic names for the four access modes.

The access mode actually used in the delete operation is determined by "maximizing" the access mode of the caller with the access mode specified by the **acmode** argument, which is to say that the less privileged of the two is used.

However, if you have **SYSNAM** privilege, the delete operation is executed at the specified access mode regardless of the caller's access mode.

If this argument is omitted or is specified as 0, the access mode of the caller is used in the delete operation.

The access mode used in the delete operation determines which tables are used and which names are deleted.

DESCRIPTION

Depending on the operation, use of **\$DELLNM** may require the calling process to have certain privilege:

- Write access to the logical name table that contains a logical name is required to delete the logical name from a shareable table
- Either delete access to the logical name table or **WRITE** access to the directory table that contains the table name is required to delete a shareable logical name table
- **SYSNAM** privilege is required to delete either a logical name or table at an inner access mode
- **GRPNAM** or **SYSPRV** privilege is required to delete a logical name from a group table
- **SYSNAM** or **SYSPRV** privilege is required to delete a logical name from a system table

System Service Descriptions

\$DELLNM

CONDITION VALUES RETURNED

SS\$_NORMAL	Successful completion.
SS\$_ACCVIO	The service cannot access the location(s) specified by one or more arguments.
SS\$_BADPARAM	One or more arguments has an invalid value, or a logical name table name was not specified.
SS\$_IVLOGNAM	The lognam argument specifies a string whose length is not in the required range of 1 through 255 characters.
SS\$_IVLOGTAB	The tabnam argument does not specify a logical name table.
SS\$_NOLOGNAM	The specified logical name table does not exist, or a logical name with an access mode equal to or less privileged than the caller's access mode does not exist in the logical name table.
SS\$_NOLOGTAB	The specified logical name table does not exist.
SS\$_NOPRIV	The caller lacks the necessary privilege to delete the logical name.
SS\$_TOOMANYLNAM	Logical name translation of the table name exceeded the allowable depth (10 translations).

System Service Descriptions

\$DELMBX

\$DELMBX—Delete Mailbox

The Delete Mailbox service marks a permanent mailbox for deletion. The actual deletion of the mailbox and of its associated logical name assignment occurs when no more I/O channels are assigned to the mailbox.

FORMAT **SY\$DELMBX** *chan*

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENT

chan

VMS Usage: **channel**
type: **word (unsigned)**
access: **read only**
mechanism: **by value**

Number of the channel assigned to the mailbox that is to be deleted. The ***chan*** is a word containing this number.

DESCRIPTION

Depending on the operation, use of \$DELMBX may require the calling process to have certain privilege:

- PRMMBX privilege is required to delete a permanent mailbox
- SHMEM privilege is required to delete a mailbox located in memory shared by multiple processors

A mailbox can be deleted only from an access mode equal to or more privileged than the access mode from which the mailbox channel was assigned.

Temporary mailboxes are automatically deleted when their reference count goes to zero.

The \$DELMBX service does not deassign the channel assigned by the caller, if any. The caller must deassign the channel with the Deassign I/O Channel (\$DASSGN) service.

System Service Descriptions

\$DELMBX

CONDITION VALUES RETURNED

SS\$_NORMAL

Service successfully completed.

SS\$_DEVNOTMBX

The specified channel is not assigned to a mailbox.

SS\$_INTERLOCK

The bit map lock for allocating mailboxes from the specified shared memory is locked by another process.

SS\$_IVCHAN

An invalid channel number was specified, that is, a channel number of 0 or a number larger than the number of channels available.

SS\$_NOPRIV

The specified channel is not assigned to a device; the process does not have the privilege to delete a permanent mailbox or a mailbox in memory shared by multiple processors; or the access mode of the caller is less privileged than the access mode from which the channel was assigned.

System Service Descriptions

\$DELPRC

\$DELPRC—Delete Process

The Delete Process service allows a process to delete itself or another process.

FORMAT **SY\$DELPRC** [*pidadr*],[*prcnam*]

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS

pidadr

VMS Usage: **process_id**
type: **longword (unsigned)**
access: **modify**
mechanism: **by reference**

Process identification (PID) of the process to be deleted. The **pidadr** argument is the address of a longword that contains the PID.

The **pidadr** argument must be specified to delete processes in other UIC groups.

prcnam

VMS Usage: **process_name**
type: **character-coded text string**
access: **read only**
mechanism: **by descriptor—fixed length string descriptor**

Process name of the process to be deleted. The **prcnam** is the address of a character string descriptor pointing to a 1- to 15-character process name string.

The **prcnam** argument can be used to delete only processes in the same UIC group as the calling process. The reason for this is that process names are unique to UIC groups, and VAX/VMS uses the UIC group number of the calling process to interpret the process name specified by the **prcnam** argument.

The **pidadr** argument must be used to delete processes in other groups.

If neither the **pidadr** nor **prcnam** arguments are specified, \$DELPRC deletes the calling process; control is not returned.

DESCRIPTION

Depending on the operation, use of \$DELPRC may require the calling process to have certain privilege.

- GROUP privilege is required to delete processes in the same group that do not have the same UIC
- WORLD privilege is required to delete any process in the system

The Delete Process system service requires system dynamic memory.

Deductible resource quotas granted to subprocesses are returned to the creator when the subprocesses are deleted.

When a process or subprocess is deleted, a termination message is sent to its creator, provided that the mailbox to receive the message still exists and the creating process has access to the mailbox. The termination message is sent before the final rundown is initiated; thus, the creator may receive the message before the process deletion is complete.

Due to the complexity of the required run-down operations, a significant time interval occurs between a delete request and the actual deletion of the process. However, the \$DELPRC service returns to the caller immediately after initiating the run-down operation.

If subsequent delete requests are issued for a process currently being deleted, the requests return immediately with a successful completion status.

For a complete list of the actions performed by the system when it deletes a process, see Section 8.7 and Section 8.6.

CONDITION VALUES RETURNED

SS\$_NORMAL	Service successfully completed.
SS\$_ACCVIO	The process name string or string descriptor cannot be read by the caller, or the process identification cannot be written by the caller.
SS\$_INSFMEM	Insufficient system dynamic memory is available to complete the operation.
SS\$_NONEXPR	Warning. The specified process does not exist, or an invalid process identification was specified.
SS\$_NOPRIV	The caller does not have the privilege to delete the specified process.

System Service Descriptions

\$DELTVA

\$DELTVA—Delete Virtual Address Space

The Delete Virtual Address Space service deletes a range of addresses from a process's virtual address space. Upon successful completion of the service, the deleted pages are inaccessible, and references to them cause access violations.

FORMAT **SY\$DELTVA** *inadr* , [*retadr*] , [*acmode*]

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS *inadr*

VMS Usage: **address_range**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Starting and ending virtual addresses of the pages to be deleted. The *inadr* argument is the address of a two-longword array containing, in order, the starting and the ending process virtual addresses. If the starting and ending virtual addresses are the same, a single page is deleted. Only the virtual page number portion of each virtual address is used; the low-order 9 bits are ignored.

The \$DELTVA service deletes pages starting at the address contained in the second longword of the *inadr* argument and ending at the address in the first longword. Thus, if the same address array is used for both the Create Virtual Address Space (\$CRETVA) and the \$DELTVA services, the pages are deleted in the reverse order from which they were created.

retadr

VMS Usage: **address_range**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Starting and ending process virtual addresses of the pages that \$DELTVA has actually deleted. The *retadr* is the address of a two-longword array containing, in order, the starting and ending process virtual addresses.

System Service Descriptions

\$DELTVA

acmode

VMS Usage: **access_mode**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Access mode on behalf of which the service is to be performed. The **acmode** argument is a longword containing the access mode.

The most privileged access mode used is the access mode of the caller. The calling process can delete pages only if those pages are owned by an access mode equal to or less privileged than the access mode of the calling process.

DESCRIPTION

If any of the pages in the specified range have already been deleted or do not exist, the service continues as if the pages were successfully deleted.

If an error occurs while pages are being deleted, the **retadr** argument specifies the pages that were successfully deleted before the error occurred. If no pages are deleted, both longwords in the return address array contain the value -1.

CONDITION VALUES RETURNED

SS\$_NORMAL

Service successfully completed.

SS\$_ACCVIO

The input address array cannot be read by the caller, or the return address array cannot be written by the caller.

SS\$_NOPRIV

A page in the specified range is in the system address space.

SS\$_PAGOWNVIO

A page in the specified range is owned by an access mode more privileged than the access mode of the caller.

System Service Descriptions

\$DEQ

\$DEQ—Dequeue Lock Request

The Dequeue Lock Request service dequeues (unlocks) granted locks; dequeues the sublocks of a lock; or cancels an ungranted lock request. The calling process must have previously acquired the lock or queued the lock request by calling the Enqueue Lock Request (\$ENQ) service.

The \$DEQ, \$ENQ, \$ENQW (Enqueue Lock Request and Wait), and \$GETLKI (Get Lock Information) services together provide the user interface to the VAX/VMS lock management facility. Refer to the descriptions of these other services and to Section 12 in Part I for additional information about lock management.

FORMAT

SY\$DEQ [*lkid*] [,*valblk*] [,*acmode*] [,*flags*]

If *lkid* is not specified, the LCK\$M_DEQALL flag in the *flags* argument must be specified.

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS

lkid
VMS Usage: **lock_id**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Lock identification of the lock to be dequeued. The *lkid* argument specifies this lock identification.

When the LCK\$M_DEQALL flag in the *flags* argument is specified, different values (or no value) for the *lkid* argument produce varying behavior:

- When *lkid* is not specified (or is specified as 0) and the LCK\$M_DEQALL flag is specified, \$DEQ dequeues all locks held by the process, at access modes equal to or less privileged than the effective access mode, on all resources. The effective access mode is the least privileged of the caller's access mode and the access mode specified in the *acmode* argument.
- When *lkid* is specified as a nonzero value and the LCK\$M_DEQALL flag is specified, \$DEQ dequeues all sublocks of the lock identified by *lkid*; it does not dequeue the lock identified by *lkid*. For this operation, \$DEQ ignores the LCK\$M_CANCEL flag if it is set. A sublock of a lock is a lock that was created by specifying the *parid* argument in the call to \$ENQ, where *parid* is the lock id of the parent lock.

The \$DEQ service returns the invalid lock id condition value (SS\$_IVLOCKID) if the *lkid* argument is omitted (or specified as 0) and the LCK\$_M_DEQALL flag is not set.

valblk

VMS Usage: **lock_value_block**
type: **longword (unsigned)**
access: **modify**
mechanism: **by reference**

Lock value block for the resource associated with the lock to be dequeued. The *valblk* argument is the address of the 16-byte lock value block. This argument is not used when the LCK\$_M_DEQALL flag is specified.

When a protected write (PW) or exclusive (EX) mode lock is being dequeued and a lock value block was specified in the *valblk* argument, the contents of that lock value block are written to the lock value block in the lock database. Further, if the lock value block in the lock database was marked as invalid, that condition is cleared; the block becomes valid.

acmode

VMS Usage: **access_mode**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Access mode of the lock to be dequeued. The *acmode* argument is a longword containing the access mode. The \$PSLDEF macro defines the following symbols for the four access modes:

Symbol	Access mode
PSL\$_KERNEL	Kernel mode
PSL\$_EXEC	Executive mode
PSL\$_SUPER	Supervisor mode
PSL\$_USER	User mode

The effective access mode used by \$DEQ is the least privileged of the caller's access mode and the access mode specified by the *acmode* argument. If *acmode* is not specified, \$DEQ uses the caller's access mode.

flags

VMS Usage: **mask_longword**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Flags specifying options for the \$DEQ operation. The *flags* argument is a longword bit mask that is the logical OR of each bit set, where each bit corresponds to an option.

A symbolic name for each flag bit is defined by the \$LCKDEF macro. The following list describes each flag.

System Service Descriptions

\$DEQ

Flag	Description
LCK\$M_DEQALL	When this flag is specified, \$DEQ dequeues multiple locks, depending on the value of the lkid argument. Refer to the description of the lkid argument for details. If LCK\$M_DEQALL is specified, the LCK\$M_CANCEL flag, if set, is ignored.
LCK\$M_CANCEL	<p>When this flag is specified, \$DEQ attempts to cancel a lock conversion request that was queued by \$ENQ. This attempt can only succeed if the lock request has not yet been granted, in which case the request is in the conversion queue. The LCK\$M_CANCEL flag is ignored if the LCK\$M_DEQALL flag is specified. Specifying the LCK\$M_CANCEL flag can result in the following actions:</p> <p>If the lock conversion request has already been granted, then the attempt to cancel the request has failed; in this case \$DEQ returns the condition value SS\$_CANCELGRANT in RO.</p> <p>If the lock conversion request has not yet been granted, \$DEQ cancels the request. As a result, the lock is regranted at its previous lock mode; the \$ENQ service receives the condition value SS\$_CANCEL in the lock status block; and the \$DEQ service returns the condition value SS\$_NORMAL in RO.</p> <p>If the lock request was not a conversion request, but was a new lock request and was therefore on the waiting queue, \$DEQ aborts the lock request. As a result, the \$ENQ service receives the condition value SS\$_ABORT in the lock status block, and \$DEQ returns the condition value SS\$_NORMAL in RO.)</p>
LCK\$M_INVVALBLK	<p>When this flag is specified, \$DEQ marks the lock value block, which is maintained for the resource in the lock database, as invalid. The lock value block remains marked as invalid until it is again written to. The Description section of the \$ENQ service provides additional information about lock value block invalidation.</p> <p>This flag is ignored if (1) the lock mode of the lock being dequeued is not protected write or exclusive, or (2) the LCK\$M_CANCEL flag is specified.</p>

DESCRIPTION

When a protected write (PW) or exclusive (EX) mode lock is being dequeued and a lock value block was specified in the **valblk** argument, the contents of that lock value block are written to the lock value block in the lock database.

If the LCK\$M_INVVALBLK flag in the **flags** argument is specified and the lock mode of the lock being dequeued is PW or EX, the lock value block in the lock database is marked as invalid whether or not a lock value block was specified in the **valblk** argument.

System Service Descriptions

\$DEQ

CONDITION VALUES RETURNED

SS\$_NORMAL

Lock was dequeued successfully.

SS\$_ACCVIO

The value block specified by the **valblk** argument cannot be accessed by the caller.

SS\$_IVLOCKID

An invalid or nonexistent lock identification was specified or the process does not have the privilege to dequeue a lock at the specified access mode.

SS\$_SUBLOCKS

The lock has sublocks and cannot be dequeued.

SS\$_CANCELGRANT

The LCK\$_M_CANCEL flag in the **flags** argument was specified, but the lock request that \$DEQ was to cancel had already been granted.

System Service Descriptions

\$DGBLSC

\$DGBLSC—Delete Global Section

The Delete Global Section service marks an existing permanent global section for deletion. The actual deletion of the global section takes place when all processes that have mapped the global section have deleted the mapped pages.

FORMAT **SYS\$DGBLSC** [*flags*] ,*gsdnam* ,*[ident]*

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS

flags

VMS Usage: **mask_longword**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Mask indicating global section characteristics. The **flags** argument is a longword value. A value of 0 (the default) specifies a group global section; a value of SEC\$M_SYSGBL specifies a system global section.

gsdnam

VMS Usage: **section_name**
type: **character-coded text string**
access: **read only**
mechanism: **by descriptor—fixed length string descriptor**

Name of the global section to be deleted. The **gsdnam** is the address of a character string descriptor pointing to this name string. Section 11.6.5.1 describes the format of this name string.

For group global sections, VAX/VMS interprets the group UIC as part of the global section name; thus, the names of global sections are unique to UIC groups.

ident

VMS Usage: **section_id**
type: **quadword (unsigned)**
access: **read only**
mechanism: **by reference**

Identification value specifying the version number of the global section to be deleted and the matching criteria to be applied. The **ident** argument is the address of a quadword structure containing three fields.

System Service Descriptions

\$DGBLSC

The version number is in the second longword. The version number contains two fields: a minor identification in the low-order 24 bits and a major identification in the high-order 8 bits. Values for these fields can be assigned by installation convention to differentiate versions of global sections. If no version number is specified when a section is created, processes that specify a version number when mapping cannot access the global section.

The first longword specifies, in its low-order 3 bits, the matching criteria. The valid values, symbolic names by which they can be specified, and their meanings are listed in the following table.

Value	Name	Match Criteria
0	SEC\$K_MATALL	Match all versions of the section
1	SEC\$K_MATEQU	Match only if major and minor identifications match
2	SEC\$K_MATLEQ	Match if the major identifications are equal and the minor identification of the mapper is less than or equal to the minor identification of the global section

If no address is specified or is specified as 0 (the default), the version number and match control fields default to 0.

DESCRIPTION

Depending on the operation, use of \$DGBLSC may require the calling process to have certain privilege:

- SYSGBL privilege is required to delete a system global section
- PRMGBL privilege is required to delete a permanent global section
- PFNMAP privilege is required to delete a page frame section
- SHMEM privilege is required to delete a global section located in memory shared by multiple processors

After a global section has been marked for deletion, any process that attempts to map it receives the warning return status code SS\$_NOSUCHSEC.

Temporary global sections are automatically deleted when the count of processes using the section goes to 0.

The \$DGBLSC service does not unmap a global section from a process's virtual address space. To accomplish this, the process should call the Delete Virtual Address Space (\$DELTVA) service, which deletes the pages to which the section is mapped.

A section located in memory that is shared by multiple processors can be marked for deletion only by a process running on the same processor that created the section.

System Service Descriptions

\$DGBLSC

CONDITION VALUES RETURNED

SS\$_NORMAL	Service successfully completed.
SS\$_ACCVIO	The global section name or name descriptor or the section identification field cannot be read by the caller.
SS\$_INTERLOCK	The bit map lock for allocating global sections from the specified shared memory is locked by another process.
SS\$_IVLOGNAM	The global section name has a length of 0 or has more than 15 characters.
SS\$_IVSECFLG	An invalid flag, reserved flag, or flag requiring a user privilege has been set.
SS\$_IVSECIDCTL	The section identification match control field is invalid.
SS\$_NOPRIV	The caller does not have the privilege to delete a system global section, does not have read/write access to a group global section, or does not have the privilege to delete a global section located in memory that is shared by multiple processors.
SS\$_NOSUCHSEC	Warning. The specified global section does not exist, or the identifications do not match.
SS\$_NOTCREATOR	The section is in memory shared by multiple processors and was created by a process on another processor.
SS\$_SHMNOTCNCT	The shared memory named in the gsdnam argument is not known to the system. This error can be caused by a spelling error in the string, an improperly assigned logical name, or the failure to identify the memory as shared at system generation time.
SS\$_TOOMANYLNAM	Logical name translation of the gsdnam string exceeded the allowed depth of 10.

\$DISMOU—Dismount Volume

The Dismount Volume service dismounts a mounted volume or volume sets.

FORMAT **SYSDISMOU** *devnam* ,*[flags]*

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS *devnam*

VMS Usage: **device_name**
type: **character-coded text string**
access: **read only**
mechanism: **by descriptor—fixed length string descriptor**

Device name of the device to be dismounted. The *devnam* argument is the address of a character string descriptor pointing to the device name string. The string may be either a physical device name or a logical name. If it is a logical name, it must translate to a physical device name.

flags

VMS Usage: **mask_longword**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

A longword bit vector specifying options for the dismount operation. The **flags** argument is a longword bit vector wherein a bit, when set, selects the corresponding option. Each bit has a symbolic name; these names are defined by the \$DMTDEF macro. The symbols and their meanings are listed below.

System Service Descriptions

\$DISMOU

Flag	Meaning
DMT\$M_ABORT	The volume is to be dismounted even if the caller did not mount the volume. If the volume was mounted with MNT\$M_SHARE specified, \$MOUNT dismounts the volume for all of the users who mounted it. To specify DMT\$M_ABORT, the caller must: (1) have GRPNAM privilege for a group volume; (2) have SYSNAM privilege for a system volume; or (3) either own the volume or have VOLPRO privilege.
DMT\$M_CLUSTER	The volume is to be dismounted cluster-wide, that is, from all nodes in the VAXcluster. \$MOUNT dismounts the volume from the caller's node first, and then from every other node in the existing VAXcluster. DMT\$M_CLUSTER dismounts only system or group volumes. To dismount a group volume cluster-wide, the caller must have GRPNAM privilege. To dismount a system volume cluster-wide, the caller must have SYSNAM privilege. DMT\$M_CLUSTER has no effect if the system is not a member of a VAXcluster. DMT\$M_CLUSTER applies only to disks.
DMT\$M_NOUNLOAD	Volume is not unloaded.
DMT\$M_UNIT	The specified device, rather than the entire volume set, is dismounted.

DESCRIPTION Depending on the operation, use of \$DISMOU may require the calling process to have certain privilege.

- GRPNAM privilege is required to dismount a volume mounted using the /GROUP qualifier
- SYSNAM privilege is required to dismount a volume mounted using the /SYSTEM qualifier

To dismount a private volume, the caller must own the volume.

When a user issues the \$DISMOU service, \$DISMOU removes the volume from the user's list of mounted volumes, deletes the logical name (if any) associated with the volume, and decrements the mount count.

If the mount count equals 0 after being decremented, \$DISMOU marks the volume for dismount. After marking the volume for dismount, \$DISMOU waits until the volume is idle before dismounting it.

If the mount count does not equal 0 after being decremented, \$DISMOU does not mark the volume for dismount (since the volume must have been mounted shared). In this case, the total effect for the issuing process is that the process is denied access to the volume and a logical name entry is deleted.

As mentioned, once marked for dismount, a volume is not actually dismounted until it is idle. A native volume is idle when no user has an open file to the volume, and a foreign volume is idle when no channels are assigned to the volume.

System Service Descriptions

\$DISMOU

Native volumes are Files-11 structured disks or ANSI-structured tapes. Foreign volumes are not Files-11 or ANSI structured.

Once a volume is actually dismounted, nonpaged pool is returned to the system. Paged pool is also returned if the volume had been mounted using the /GROUP or /SYSTEM qualifiers.

If a volume is part of a Files-11 volume set and the flag bit DMT\$V_UNIT is not set, the entire volume set will be dismounted.

When a Files-11 volume has been marked for dismount, new channels can be assigned to the volume, but no new files can be opened.

Note that the SS\$_NORMAL status code indicates only that \$DISMOU has successfully performed one or more of the steps described above: decremented the mount count, marked the volume for dismount, or dismounted the volume. The only way to determine absolutely that the dismount has actually occurred is to check the device characteristics using the Get Device/Volume Information (\$GETDVI) service.

By specifying the DVI\$_DEVCHAR item code in a call to \$GETDVI, you can learn whether a volume is mounted (it is if the DEV\$V_MNT bit is set) or whether it is marked for dismount (it is if the DEV\$_DMT bit is set). The mount count will be zero if DEV\$V_MNT is clear or if DEV\$_DMT is set.

CONDITION VALUES RETURNED

SS\$_NORMAL	Service successfully completed.
SS\$_ACCPIO	The device name descriptor cannot be read or does not describe a readable device name.
SS\$_DEVALLOC	The device is allocated to another process and cannot be dismounted by the caller.
SS\$_IVLOGNAM	The device logical name has a length of zero or is longer than the allowable logical name length.
SS\$_IVDEVNAM	Device name string is not valid.
SS\$_NOGRPNAM	GRPNAM privilege is required to dismount a volume mounted for group-wide access.
SS\$_NOIOCHAN	No I/O channel is available. To use \$DISMOU, a channel must be assigned to the volume.
SS\$_NONLOCAL	Warning. The device is on a remote node.
SS\$_NOSUCHDEV	The specified device does not exist.
SS\$_NOSYSNAM	SYSNAM privilege is required to dismount a volume mounted for system-wide access.
SS\$_NOTFILEDEV	The specified device is not file-structured.
SS\$_DEVOFFLINE	The specified device is not available.
SS\$_DEVNOTMOUNT	The specified device is not mounted.

System Service Descriptions

\$DLCEFC

\$DLCEFC—Delete Common Event Flag Cluster

The Delete Common Event Flag Cluster service marks a permanent common event flag cluster for deletion. The cluster is actually deleted when no more processes are associated with it.

FORMAT **SY\$DLCEFC** *name*

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENT *name*

VMS Usage: **ef_cluster_name**
type: **character-coded text string**
access: **read only**
mechanism: **by descriptor—fixed length string descriptor**

Name of the common event flag cluster to be deleted. The **name** argument is the address of a character string descriptor pointing to the name of the cluster.

Refer to Section 4.7.1 for the format of this argument. The names of event flag clusters are unique to UIC groups, and the UIC group number of the calling process is part of the name.

DESCRIPTION To delete a common event flag cluster, the calling process must either have PRMCEB privilege or have the same UIC as the process that created the cluster.

The calling process must have SHMEM privilege to delete a common event flag cluster from memory that is shared by multiple processors.

The \$DLCEFC service does not disassociate a process from a common event flag cluster; the Disassociate Common Event Flag Cluster (\$DACEFC) service accomplishes this. However, the system disassociates a process from an event flag cluster at image exit.

If the cluster has already been deleted or does not exist, the \$DLCEFC service returns the status code SS\$_NORMAL.

System Service Descriptions

\$DLCEFC

CONDITION VALUES RETURNED

SS\$_NORMAL
SS\$_IVLOGNAM
SS\$_NOPRIV

Service successfully completed.

The cluster name string has a length of 0 or has more than 15 characters.

The process does not have the privilege to delete a permanent common event flag cluster, or the process does not have the privilege to delete a common event flag cluster in memory shared by multiple processors.

System Service Descriptions

\$ENQ

\$ENQ—Enqueue Lock Request

The Enqueue Lock Request service queues a new lock or lock conversion on a resource.

The \$ENQ service completes asynchronously; that is, it returns to the caller after queuing the lock request, without waiting for the lock to be either granted or converted.

For synchronous completion, use the Enqueue Lock Request and Wait (\$ENQW) service. The \$ENQW service is identical to the \$ENQ service in every way except that \$ENQW returns to the caller when the lock is either granted or converted.

For additional information about system service completion, refer to the Synchronize (\$SYNCH) service and to Section 2.5.

The \$ENQ, \$ENQW, \$DEQ (Dequeue Lock Request), and \$GETLKI (Get Lock Information) services together provide the user interface to the VAX/VMS lock management facility. Refer to the descriptions of these other services and to Section 12 in Part I for additional information about lock management.

FORMAT	SY\$ENQ <i>[efn] ,lkmode ,lksb ,[flags] ,[resnam] ,[parid] ,[astadr] ,[astprm] ,[blkast] ,[acmode] ,nullarg</i>
---------------	--

RETURNS	VMS Usage: cond_value type: longword (unsigned) access: write only mechanism: by value
----------------	---

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS	efn VMS Usage: ef_number type: longword (unsigned) access: read only mechanism: by value
------------------	---

Number of the event flag to be set when the lock request has been granted. The **efn** argument is a longword containing this number.

Upon request initiation, \$ENQ clears the specified event flag (or event flag 0 if **efn** was not specified). Then when the lock request is granted, the specified event flag (or event flag 0) is set unless the LCK\$M_SYNCSTS flag in the **flags** argument was specified.

lkmode

VMS Usage: **longword_unsigned**
 type: **longword (unsigned)**
 access: **read only**
 mechanism: **by value**

Lock mode requested. The **lkmode** argument is a longword specifying this lock mode.

Each lock mode has a symbolic name. These symbolic names are defined by the \$LCKDEF macro. The following list gives the symbolic name and description for each lock mode:

Lock Mode	Description
LCK\$K_NLMODE	Null mode. This mode grants no access to the resource but serves rather as a place holder and indicator of future interest in the resource. The null mode does not inhibit locking at other lock modes; further, it prevents the deletion of the resource and lock value block, which would otherwise occur if the locks held at the other lock modes were dequeued.
LCK\$K_CRMODE	Concurrent read. This mode grants the caller read access to the resource while permitting write access to the resource by other users. This mode is used to read data from a resource in an unprotected manner, since other users could modify that data as it was being read. This mode is typically used when additional locking is being performed at a finer granularity with sublocks.
LCK\$K_CWMODE	Concurrent write. This mode grants the caller write access to the resource while permitting write access to the resource by other users. This mode is used to write data to a resource in an unprotected fashion, since other users may simultaneously write data to the resource. This mode is typically used when additional locking is being performed at a finer granularity with sublocks.
LCK\$K_PRMODE	Protected read. This mode grants the caller read access to the resource while permitting only read access to the resource by other users. Write access is not allowed. This is the traditional "share lock."
LCK\$K_PWMODE	Protected write. This mode grants the caller write access to the resource while permitting only read access to the resource by other users; the other users must have specified concurrent read mode access. No other writers are allowed access to the resource. This is the traditional "update lock."
LCK\$K_EXMODE	Exclusive. The exclusive mode grants the caller write access to the resource and allows no access to the resource by other users. This is the traditional "exclusive lock."

lksb

VMS Usage: **lock_status_block**
 type: **longword (unsigned)**
 access: **write only**

System Service Descriptions

\$ENQ

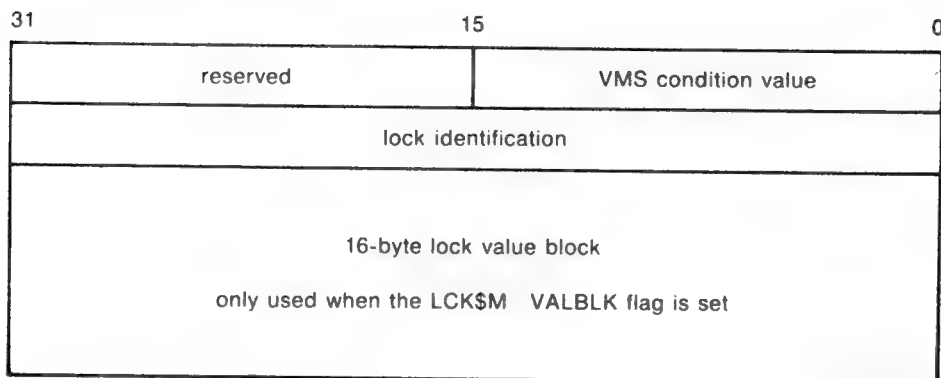
mechanism: **by reference**

Lock status block in which \$ENQ writes the final completion status of the operation. The **lksb** argument is the address of the 8-byte lock status block.

The lock status block may optionally contain a 16-byte lock value block.

When the LCK\$M_VALBLK flag is specified in the **flags** argument, the lock status block contains a lock value block; in this case, the 16-byte lock value block appears beginning at the first byte following the eighth byte of the lock status block, bringing the total length of the lock status block to 24 bytes.

The diagram below shows the format of the lock status block, showing the optional lock value block.



ZK-1708-84

Lock Status Block Fields

Condition value

A word in which \$ENQ writes a VAX/VMS condition value describing the final disposition of the lock request, for example, whether the lock was granted, converted, and so on. The condition values returned in this field are described under the heading "Condition Values Returned in the Lock Status Block," which appears following the list of condition values returned in R0.

Reserved

A word that is reserved to DIGITAL.

Lock identification

A longword containing the lock identification of the lock.

For a new lock, \$ENQ writes the lock identification of the requested lock into this longword when the lock request is queued.

For a lock conversion on an existing lock, the user must supply the lock identification of the existing lock in this field.

Lock value block

A user-defined, 16-byte structure containing information about the resource. This information is user defined and is interpreted only by the user program.

When a process acquires a lock on a resource, the lock management facility provides that process with a process-private copy of the lock value block associated with the resource, providing that process has specified the LCK\$M_VALBLK flag in the **flags** argument. The copy provided to the process is a copy of the lock value block stored in the lock manager's data base.

The copy of the lock value block maintained in the lock database is updated in the following way: whenever a process either (1) dequeues a lock at protected write (PW) or exclusive (EX) mode or (2) converts a lock at one of these modes to a lower lock mode, VAX/VMS stores the caller's lock value block in the lock database, providing the caller has specified the LCK\$_M_VALBLK flag.

Callers of \$ENQ are provided with copies of the updated lock value block from the lock database in the following way: when \$ENQ grants a new lock to the caller or converts the caller's existing lock to a higher lock mode, \$ENQ copies the lock value block from the lock database to the caller's lock value block, providing the caller has specified the LCK\$_M_VALBLK flag.

The Description section describes events that may cause the lock value block to become invalid.

flags

VMS Usage: **mask_longword**
 type: **longword (unsigned)**
 access: **read only**
 mechanism: **by value**

Flags specifying options for the \$ENQ operation. The **flags** argument is a longword bit mask that is the logical OR of each bit set, where each bit corresponds to an option.

A symbolic name for each flag bit is defined by the \$LCKDEF macro. The following list describes each flag.

Flag	Description
LCK\$_M_NOQUEUE	When this flag is specified, \$ENQ does not queue the lock request unless the lock can be granted immediately. By default, \$ENQ will always queue the request. If LCK\$_M_NOQUEUE is specified in a lock conversion operation and the conversion cannot be granted immediately, the lock remains in the original lock mode.
LCK\$_M_SYNCSTS	When this flag is specified, \$ENQ returns the successful condition value SS\$_SYNCH in R0 if the lock request is granted immediately; in this case, no completion AST is delivered and no event flag is set. If the lock request is queued successfully but cannot be granted immediately, \$ENQ returns the condition value SS\$_NORMAL in R0; then when the request is granted, \$ENQ sets the event flag and queues an AST if the astadr argument was specified.
LCK\$_M_SYSTEM	When this flag is specified, the resource name is interpreted as system-wide. By default, resource names are qualified by the UIC group number of the creating process. This flag is ignored in lock conversions.
LCK\$_M_VALBLK	When this flag is specified, the lock status block contains a lock value block. See the description of the lksb argument for more information.

System Service Descriptions

\$ENQ

Flag	Description
LCK\$M_CONVERT	When this flag is specified, \$ENQ performs a lock conversion. In this case, the caller must supply (in the second longword of the lock status block) the lock identification of the lock to be converted.
LCK\$M_NODLCKWT	<p>By specifying this flag, a process indicates to the lock management services that it is not blocked from execution while waiting for the lock request to complete. An example of this is a situation where a lock request is left outstanding on the waiting queue as a signaling device between processes.</p> <p>This flag helps to prevent false deadlocks by providing the lock management services with additional information about the process issuing the lock request. When this flag is set, the lock management services do not consider this lock when trying to detect deadlock conditions.</p> <p>A process should only specify the LCK\$M_NODLCKWT flag in a call to the \$ENQ system service. The \$ENQW system service waits for the lock request to be granted before returning to the caller; therefore, specifying the LCK\$M_NODLCKWT flag in a call to the \$ENQW system service defeats the purpose of the flag and can result in a genuine deadlock being ignored.</p> <p>The lock management services make use of the LCK\$M_NODLCKWT flag only when the lock specified by the call to \$ENQ is in either the waiting or the conversion queue.</p> <p>Improper use of the LCK\$M_NODLCKWT flag can result in the lock management services ignoring genuine deadlocks.</p>
LCK\$M_NODLCKBLK	<p>By specifying this flag, a process indicates to the lock management services that, if this lock is blocking another lock request, the process intends to give up this lock on demand. When this flag is specified, the lock management services do not consider this lock as blocking other locks when trying to detect deadlock conditions.</p> <p>A process typically specifies the LCK\$M_NODLCKBLK flag only when it also specifies a blocking AST. Blocking ASTs notify processes with granted locks that another process with an incompatible lock mode has been queued to access the same resource. Use of blocking ASTs may cause false deadlocks, because the lock management services detect a blocking condition, even though a blocking AST has been specified; however, the blocking condition will disappear as soon as the process holding the lock executes, receives the blocking AST and dequeues the lock. Specifying the LCK\$M_NODLCKBLK flag prevents this type of false deadlock.</p>

System Service Descriptions

\$ENQ

Flag	Description
	<p>To enable blocking ASTs, the blkast argument of the \$ENQ system service must contain the address of a blocking AST service routine. If the process specifies the LCK\$M_NODLCKBLK flag, the blocking AST service routine should either dequeue the lock or convert it to a lower lock mode without issuing any new lock requests. If the blocking AST routine does otherwise, a genuine deadlock could be ignored.</p> <p>The lock management services make use of the LCK\$M_NODLCKBLK flag only when the lock specified by the call to \$ENQ has been granted.</p> <p>Improper use of the LCK\$M_NODLCKBLK flag can result in the lock management services ignoring genuine deadlocks.</p>
LCK\$M_NOQUOTA	<p>This flag is reserved to DIGITAL. When this flag is set, the calling process is not charged Enqueue Limit (ENQLM) quota for this new lock. The calling process must be running in executive or kernel mode to set this flag. This flag is ignored for lock conversions.</p>
LCK\$M_CVTSYS	<p>This flag is reserved to DIGITAL. When this flag is set, the lock is converted from a process-owned lock to a system-owned lock. The calling process must be running in executive or kernel mode to set this flag.</p>

resnam

VMS Usage: **char_string**
 type: **character-coded text string**
 access: **read only**
 mechanism: **by descriptor—fixed length string descriptor**

Name of the resource to be locked by this lock. The **resnam** argument is the address of a character string descriptor pointing to this name. The name string may be from 1 to 31 bytes in length.

The **resnam** argument is required for new locks and is ignored for lock conversions.

parid

VMS Usage: **lock_id**
 type: **longword (unsigned)**
 access: **read only**
 mechanism: **by value**

Lock identification of the parent lock. The **parid** argument is a longword containing this identification value.

If this argument is not specified or is specified as 0, \$ENQ assumes that the lock does not have a parent lock. This argument is optional for new locks and is ignored for lock conversions.

astadr

VMS Usage: **ast_procedure**
 type: **procedure entry mask**
 access: **call without stack unwinding**

System Service Descriptions

\$ENQ

mechanism: **by reference**

AST service routine to be executed when the lock is either granted or converted. The **astadr** argument is the address of the entry mask of this routine.

If **astadr** is specified, the AST routine will execute at the same access mode as the caller of \$ENQ.

astprm

VMS Usage: **user_arg**

type: **longword (unsigned)**

access: **read only**

mechanism: **by value**

AST parameter to be passed to the AST routine specified by the **astadr** argument. The **astprm** argument specifies this longword parameter.

blkast

VMS Usage: **ast_procedure**

type: **procedure entry mask**

access: **call without stack unwinding**

mechanism: **by reference**

Blocking AST routine to be called whenever this lock is granted and is blocking any other lock requests. The **blkast** argument is the address of the entry mask to this routine.

A parameter may be passed to this routine by using the **astprm** argument.

acmode

VMS Usage: **access_mode**

type: **longword (unsigned)**

access: **read only**

mechanism: **by value**

Access mode to be associated with the lock. The **acmode** argument is a longword containing the access mode. The \$PSLDEF macro defines the following symbols for the four access modes:

Symbol	Access mode
PSL\$C_KERNEL	Kernel mode
PSL\$C_EXEC	Executive mode
PSL\$C_SUPER	Supervisor mode
PSL\$C_USER	User mode

The \$ENQ service associates an access mode with the lock in the following way:

- If a parent lock was specified (by the **parid** argument), \$ENQ uses the access mode associated with the parent lock and ignores both the **acmode** argument and the caller's access mode.
- If the lock has no parent lock (the **parid** argument was not specified or was specified as 0), \$ENQ uses the least privileged of the caller's access mode and the access mode specified by the **acmode** argument. If **acmode** was not specified, \$ENQ uses the caller's access mode.

nullarg

VMS Usage: **null_arg**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Place-holding argument. This argument is reserved to DIGITAL.

DESCRIPTION

To queue a lock on a system-wide resource, the calling process must either have SYSLCK privilege or be executing in executive or kernel mode.

To specify a parent lock when queuing a lock, the access mode associated with the parent lock must be equal to, or less privileged than, the access mode of the calling process.

To queue a lock conversion, the access mode associated with the lock being converted must be equal to, or less privileged than, the access mode of the calling process.

\$ENQ uses the following system resources:

- The \$ENQ service uses the Enqueue Limit (ENQLM) quota
- The \$ENQ service may use AST limit (ASTLM) quota in lock conversion requests that specify either the **astadr** or **blkast** arguments
- System dynamic memory is required for the creation of the lock and resource blocks

When \$ENQ queues a lock request, it returns the status of the request in R0 and writes the lock identification of the lock in the lock status block. Then when the lock request is granted, \$ENQ writes the final completion status in the lock status block, sets the event flag, and calls the AST routine, if this has been requested.

When \$ENQW queues a lock request, it returns status in R0 and in the lock status block when the lock has been either granted or converted. At this time, it also sets the event flag and calls the AST routine, if this has been requested.

Invalidation of the Lock Value Block

In some situations, the lock value block may become invalid. In these situations, \$ENQ warns the caller by returning the condition value **SS\$_VALNOTVALID** in the lock status block, providing the caller has specified the flag **LCK\$_M_VALBLK** in the **flags** argument.

The **SS\$_VALNOTVALID** condition value is a warning message, not an error message. The \$ENQ service will proceed to grant the requested lock despite the fact that it is returning **SS\$_VALNOTVALID**. Further, \$ENQ will return this warning on all subsequent calls to \$ENQ until either a new lock value block is written to the lock database or the resource is deleted. Resource deletion occurs when no locks are associated with the resource.

The following events may cause the lock value block to become invalid:

- If any process holding a protected write or exclusive mode lock on a resource is terminated abnormally, the lock value block becomes invalid.
- If a VAX node in a VAXcluster fails and a process on that node was holding (or may have been holding) a protected write or exclusive mode lock on the resource, the lock value block becomes invalid.

System Service Descriptions

\$ENQ

- If a process holding a protected write or exclusive mode lock on the resource calls the Dequeue Lock Request (\$DEQ) service to dequeue this lock and specifies the flag LCK\$M_INVVALBLK in the **flags** argument, the lock value block maintained in the lock database is marked invalid.

CONDITION VALUES RETURNED

SS\$_NORMAL	Service successfully completed; the lock request was successfully queued.
SS\$_SYNCH	Service successfully completed; the LCK\$M_SYNCSTS flag in the flags argument was specified, and \$ENQ was able to grant the lock request immediately.
SS\$_ACCVIO	The lock status block or the resource name cannot be read.
SS\$_BADPARAM	An invalid lock mode was specified in the lkmode argument.
SS\$_CVTUNGRANT	A lock conversion was attempted on a lock that is not currently granted.
SS\$_EXDEPTH	The limit of levels of sublocks has been exceeded.
SS\$_EXENQLM	The process has exceeded its Enqueue Limit (ENQLM) quota.
SS\$_INSFMEM	Insufficient system dynamic memory is available to create the necessary data structures.
SS\$_IVBUFLEN	The length of the resource name was either 0 or greater than 31.
SS\$_JVLOCKID	An invalid or nonexistent lock identification was specified, or the lock identified by the lock identification has an associated access mode that is more privileged than the caller's.
SS\$_NOLOCKID	No lock identification was available for the lock request.
SS\$_NOTQUEUED	The lock request was not queued; the LCK\$M_NOQUEUE flag in the flags argument was specified and \$ENQ was not able to grant the lock request immediately.
SS\$_NOSYSLCK	The LCK\$M_SYSTEM flag in the flags argument was specified but the caller lacks the necessary SYSLCK privilege.
SS\$_PARNOTGRANT	The parent lock specified in the parid argument was not granted.

CONDITION VALUES RETURNED IN THE LOCK STATUS BLOCK

SS\$_NORMAL	Successful completion; the lock was successfully granted or converted.
SS\$_ABORT	The lock was dequeued (by the \$DEQ service) before \$ENQ could grant the lock.
SS\$_DEADLOCK	A deadlock was detected.

System Service Descriptions

\$ENQ

SS\$_CANCEL

The lock conversion request has been canceled and the lock has been regranted at its previous lock mode. This condition value is returned when \$ENQ queues a lock conversion request, the request has not been granted yet (it is in the conversion queue), and, in the interim, the \$DEQ service is called (with the LCK\$_CANCEL flag specified) to cancel this lock conversion request. If the lock is granted before \$DEQ can cancel the conversion request, the call to \$DEQ receives the condition value SS\$_CANCELGRANT, and the call to \$ENQ receives SS\$_NORMAL.

SS\$_VALNOTVALID

The lock value block is marked as invalid. This warning message is returned only if the caller has specified the flag LCK\$_VALBLK in the **flags** argument. Note that the lock has been successfully granted despite the fact that this warning message is returned. Refer to the Description section for a complete discussion of lock value block invalidation.

System Service Descriptions

\$ENQW

\$ENQW—Enqueue Lock Request and Wait

The Enqueue Lock Request and Wait service queues a lock on a resource.

The \$ENQW service completes synchronously; that is, it returns to the caller when the lock has been either granted or converted.

For asynchronous completion, use the Enqueue Lock Request (\$ENQ) service; \$ENQ returns to the caller after queuing the lock request, without waiting for the lock to be either granted or converted.

In all other respects, \$ENQW is identical to \$ENQ. Refer to the documentation of \$ENQ for all other information about the \$ENQW service.

For additional information about system service completion, refer to the documentation of the Synchronize (\$SYNCH) service and to Section 2.5.

The \$ENQ, \$ENQW, \$DEQ (Dequeue Lock Request), and \$GETLKI (Get Lock Information) services together provide the user interface to the VAX/VMS lock management facility. Refer to the descriptions of these other services and to Section 12 in Part I for additional information about lock management.

FORMAT

SY\$ENQW *[efn] ,lkmode ,lksb ,[flags] ,[resnam]
 ,[parid] ,[astadr] ,[astprm] ,[blkast]
 ,[acmode] ,nullarg*

\$ERAPAT—Get Security Erase Pattern

The Get Security Erase Pattern service generates a security erase pattern. A user-written erase routine can then write this pattern into areas of memory containing sensitive data that is no longer in use to prevent the inadvertent disclosure of the sensitive data.

FORMAT **SY\$ERAPAT** *[type],[count],[patadr]*

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED".

ARGUMENTS *type*

VMS Usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Type of storage to be written over with the erase pattern. The **type** argument is a longword containing the type of storage. The three storage types and their symbolic names (defined by the \$ERADEF macro) follow.

Storage Type	Symbolic Name
Main memory	ERA\$K_MEMORY
Disk	ERA\$K_DISK
Tape	ERA\$K_TAPE

count

VMS Usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

The number of times that \$ERAPAT has been called in a single security erase operation. The **count** argument is a longword containing the iteration count.

The \$ERAPAT service should be called initially with the **count** argument set to 1, the second time with the **count** argument set to 2, and so on, until the status code SS\$_NOTRAN is returned.

System Service Descriptions

\$ERAPAT

patadr

VMS Usage: **longword_unsigned**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

The security erase pattern to be written. The *patadr* is the address of a longword into which the security erase pattern is to be written.

DESCRIPTION

The \$ERAPAT service provides a consistent mechanism for performing security erase operations. This service is used primarily by VAX/VMS, but it may also be used by users who want to perform security erase operations on foreign disks.

The \$ERAPAT service should be called iteratively until the completion status `SS$_NOTRAN` is returned.

CONDITION VALUES RETURNED

<code>SS\$_NORMAL</code>	Successful completion; proceed with the next erase step.
<code>SS\$_NOTRAN</code>	Successful completion; security erase completed.
<code>SS\$_BADPARAM</code>	Invalid type argument or invalid count argument.
<code>SS\$_ACCVIO</code>	The <i>patadr</i> argument cannot be written by the caller.

EXAMPLE

```
; Code fragment that erases 20 blocks (blocks 15 through 34)
; on a disk
;
PATTERN:
    .LONG    0                ; Cell to contain output
                                ; from $ERAPAT
    .
    MOVL     #1, R2            ; Set initial count
    $ERADEF                                     ; Macro to define names
                                ; used by $ERAPAT
10$:  $ERAPAT_S                ; Call the $ERAPAT
        COUNT=R2,-            ; service
        TYPE=$ERA$K_DISK,-
        PATADR=PATTERN
    BLBC     R0, EXIT          ; Branch if error
    CMPL     $SS$_NOTRAN, R0    ; Are we done?
    BEQL     EXIT              ; Branch if so
    $QIO_S   FUNC=$QIO_WRITEBLK!IO$M_ERASE,- ; Call
        P1=PATTERN,-          ; to the $QIO service
        P2=<20*512>,-         ; to write the erase
        P3=#15                ; pattern into memory
    INCL     R2                ; Increase count
    BRB      10$
EXIT:  .
```

System Service Descriptions

\$ERAPAT

This example demonstrates how to use the \$ERAPAT service to perform a security erase to a disk. Note that after each call to \$ERAPAT, a test for the status SS\$_NOTRAN is made. If SS\$_NOTRAN has not been returned, \$QIO is called to write the pattern returned by \$ERAPAT onto the disk. After this write, \$ERAPAT is called again and the cycle is repeated until the code SS\$_NOTRAN is returned, at which point the security erase procedure is complete.

System Service Descriptions

\$EXIT

\$EXIT—Exit

The Exit service is used by the operating system to initiate image run-down when the current image in a process completes execution. Control normally returns to the command interpreter.

FORMAT **SYS\$EXIT** [*code*]

ARGUMENT

code

VMS Usage: **cond_value**

type: **longword (unsigned)**

access: **read only**

mechanism: **by value**

Longword value to be saved in the process header as the completion status of the current image. If not specified in a macro call, a value of 1 is passed as the completion code for VAX MACRO and VAX BLISS-32 and, a value of 0 is passed for other languages. This value can be tested at the command level to provide conditional command execution.

DESCRIPTION

The \$EXIT service is unlike all other system services in that it does not return status codes in R0 or anywhere else. The \$EXIT service does not return control to the caller; it performs an exit to the command interpreter or causes the process to terminate if no command interpreter is present.

For a summary of the actions taken by the system at image exit, see Section 8.6.

**CONDITION
VALUES
RETURNED**

None

\$EXPREG—Expand Program/Control Region

The Expand Program/Control Region service adds a specified number of new virtual pages to a process's program region or control region for the execution of the current image. Expansion occurs at the current end of that region's virtual address space.

FORMAT **SYS\$EXPREG** *pagcnt* ,*[retadr]* ,*[acmode]* ,*[region]*

RETURNS

VMS Usage: **cond_value**
 type: **longword (unsigned)**
 access: **write only**
 mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS

pagcnt

VMS Usage: **longword_unsigned**
 type: **longword (unsigned)**
 access: **read only**
 mechanism: **by value**

Number of pages to add to the current end of the program or control region. The ***pagcnt*** argument is a longword value containing this number.

retadr

VMS Usage: **address_range**
 type: **longword (unsigned)**
 access: **write only**
 mechanism: **by reference**

Starting and ending process virtual addresses of the pages that \$EXPREG has actually added. The ***retadr*** is the address of a two-longword array containing, in order, the starting and ending process virtual addresses.

acmode

VMS Usage: **access_mode**
 type: **longword (unsigned)**
 access: **read only**
 mechanism: **by value**

Access mode to be associated with the newly added pages. The ***acmode*** argument is a longword containing the access mode.

The most privileged access mode used is the access mode of the caller.

The newly added pages are given the following protection: (1) read and write access for access modes equal to or more privileged than the access mode used in the call and (2) no access for access modes less privileged than that used in the call.

System Service Descriptions

\$EXPREG

region

VMS Usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Number specifying which program region is to be expanded. The **region** argument is a longword value. A value of 0 (the default) specifies that the program region (P0 region) is to be expanded. A value of 1 specifies that the control region (P1 region) is to be expanded.

DESCRIPTION

The process's paging file quota (PGFLQUOTA) must be sufficient to accommodate the increased size of the virtual address space.

The new pages, which were previously inaccessible to the process, are created as demand-zero pages.

Because the bottom of the user stack is normally located at the end of the control region, expanding the control region is equivalent to expanding the user stack. The effect is to increase the available stack space by the specified number of pages.

The starting address returned is always the first available page in the designated region; therefore, the ending address is smaller than the starting address when the control region is expanded and is larger than the starting address when the program region is expanded.

If an error occurs while adding pages, the **retadr** argument (if specified) indicates the pages that were successfully added before the error occurred. If no pages were added, both longwords of the **retadr** argument contain the value -1.

The information returned in the location addressed by the **retadr** argument (if specified) can be used as the input range to the Delete Virtual Address Space (\$DELTVA) service.

CONDITION VALUES RETURNED

SS\$_NORMAL	Service successfully completed.
SS\$_ACCVIO	The return address array cannot be written by the caller.
SS\$_EXQUOTA	The process exceeded its paging file quota.
SS\$_ILLPAGCNT	The specified page count was less than 1.
SS\$_INSFWSL	The process's working set limit is not large enough to accommodate the increased virtual address space.
SS\$_VASFULL	The process's virtual address space is full. No space is available in the process page table for the requested regions.

\$FAO—Formatted ASCII Output

The Formatted ASCII Output service (1) converts a binary value into an ASCII character string in decimal, hexadecimal, or octal notation and returns the character string in an output string and (2) inserts variable character string data into an output string.

The Formatted ASCII Output with List Parameter (\$FAOL) service provides an alternate way to specify input parameters for a call to the \$FAO system service. The formats for both \$FAO and \$FAOL appear below.

FORMAT

SYSS\$FAO *ctrstr*,[*outlen*],*outbuf*,[*p1*]...[*Pn*]
SYSS\$FAOL *ctrstr*,[*outlen*],*outbuf*,[*prmlst*]

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS

ctrstr

VMS Usage: **char_string**
type: **character-coded text string**
access: **read only**
mechanism: **by descriptor—fixed length string descriptor**

Control string passed to \$FAO. The *ctrstr* argument is the address of a character string descriptor pointing to the control string. The control string contains the text to be output together with one or more FAO directives, which are described in the Description section.

There is no restriction on the length of the control string, nor on the number of FAO directives it may contain. However, if an exclamation point (!) must appear in the output string, it must be represented in the control string by a double exclamation point (!!). A single exclamation point in the control string indicates to \$FAO that the next characters are to be interpreted as FAO directives.

When \$FAO processes the control string, it writes each character that is not part of an FAO directive to the output buffer. When it encounters an exclamation point, it interprets the following characters as an FAO directive.

If the FAO directive is valid, \$FAO processes it. If the directive requires a parameter, \$FAO processes the next consecutive parameter in the specified parameter list. If the FAO directive is not valid, \$FAO terminates and returns a condition value in R0.

System Service Descriptions

\$FAO

The \$FAO service reads parameters from the argument list specified in the call; these arguments have the names p1, p2, p3, and so on, up to p20. Each argument specifies one parameter. Since \$FAO accepts a maximum of 20 parameters in a single call, the \$FAOL service must be used if the number of parameters exceeds 20. The \$FAOL service accepts any number of parameters by using the **prmlst** argument.

outlen

VMS Usage: **word_unsigned**
type: **word (unsigned)**
access: **write only**
mechanism: **by reference**

Length in bytes of the fully formatted output string returned by \$FAO. The **outlen** argument is the address of a word containing this value.

outbuf

VMS Usage: **char_string**
type: **character-coded text string**
access: **write only**
mechanism: **by descriptor—fixed length string descriptor**

Output buffer into which \$FAO writes the fully formatted output string. The **outbuf** argument is the address of a character string descriptor pointing to the output buffer.

p1 to pn

VMS Usage: **varying_arg**
type: **longword (signed)**
access: **read only**
mechanism: **by value**

FAO directive parameter(s). The **p1** argument is a longword containing the parameter needed by the first FAO directive encountered in the control string, the **p2** argument is a longword containing the parameter needed for the second FAO directive, and so on for the remaining arguments up to and including **p20**. If an FAO directive does not require a parameter, that FAO directive is processed without reading a parameter from the argument list.

Depending on the directive, a parameter may be: a value that is to be converted, an address of a string to be inserted into the output string, or a length or argument count. Each directive in the control string may require a corresponding parameter or parameters.

prmlst

VMS Usage: **vector_longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

List of FAO directive parameters to be passed to the \$FAOL service. The **prmlst** argument is the address of a list of longwords wherein each longword is a parameter. \$FAOL processes these parameters sequentially as it encounters, in the control string, FAO directives that require parameters.

The parameter list may be a data structure that already exists in a program and from which certain values are to be extracted.

DESCRIPTION

The \$FAO_S macro form uses a PUSH instruction for all parameters (P1 through Pn) passed to the service; if a symbolic address is specified, it must be preceded with a number sign character (#) or loaded into a register.

A maximum of 20 parameters can be specified on the \$FAO macro. If more than 20 parameters are required, use the \$FAOL macro.

This service does not check the length of the argument list, and therefore cannot return the SS\$_INSFARG (insufficient arguments) error status code. If the service does not receive enough arguments (for example, if you omit required commas in the call), you might not get the desired result.

Format of FAO Directives

FAO directives may appear anywhere in the control string. The general format of an FAO directive is as follows:

!DD

The exclamation point (!) specifies that the following character(s) are to be interpreted as an FAO directive and the characters "DD" represent a 1- or 2-character FAO directive. When the characters of the FAO directive are alphabetic, they must be uppercase.

An FAO directive may optionally specify:

- A repeat count. The format is:

!n(DD)

In this case "n" is a decimal value specifying the number of times that \$FAO is to repeat the directive. If the directive requires a parameter or parameters, \$FAO uses successive parameters from the parameter list for each repetition of the directive; it does not use the same parameter(s) for each repetition. The parentheses are required syntax.

- An output field length. The format is:

!mDD

In this case "m" is a decimal value specifying the length of the field (within the output string) into which \$FAO is to write the output resulting from the directive. The length is expressed as a number of characters.

- Both a repeat count and output field length. In this case the format is:

!n(mDD)

Specifying Variables for Repeat Count and Field Length

Repeat counts and output field lengths may be specified as variables by using a number sign (#) in place of an absolute numeric value. If a number sign (#) is specified for a repeat count, the next parameter passed to FAO must contain the count. If a number sign (#) is specified for an output field length, the next parameter must contain the length value.

If number sign (#) is specified for both the output field length and for the repeat count, only one length parameter is required; each output string will have the specified length.

If number sign (#) is specified for the repeat count and/or for the output field length, the parameter(s) specifying the count and/or length must precede other parameters required by the directive.

System Service Descriptions

\$FAO

Table SYS-2 List of FAO Directives

Directive	Description
Directives for Character String Substitution	
IAC	Inserts a counted ASCII string. It requires one parameter: the address of the string to be inserted. The first byte of the string must contain the length in characters of the string.
IAD	Inserts an ASCII string. It requires two parameters: the length of the string and the address of the string. Each of these parameters is a separate argument.
IAF	Inserts an ASCII string and replaces all nonprintable ASCII codes with periods (.). It requires two parameters: the length of the string and the address of the string. Each of these parameters is a separate argument.
IAS	Inserts an ASCIID string. It requires one parameter: the address of a character string descriptor pointing to the string.
Directives for Zero-Filled Numeric Conversion	
IOB	Converts a byte value to the ASCII representation of the value's octal equivalent. It requires one parameter: the value to be converted. \$FAO uses only the low-order byte of the longword parameter.
IOW	Converts a word value to the ASCII representation of the value's octal equivalent. It requires one parameter: the value to be converted. \$FAO uses only the low-order word of the longword parameter.
IOL	Converts a longword value to the ASCII representation of the value's octal equivalent. It requires one parameter: the value to be converted.
IXB	Converts a byte value to the ASCII representation of the value's hexadecimal equivalent. It requires one parameter: the value to be converted. \$FAO uses only the low-order byte of the longword parameter.
IXW	Converts a word value to the ASCII representation of the value's hexadecimal equivalent. It requires one parameter: the value to be converted. \$FAO uses only the low-order word of the longword parameter.
IXL	Converts a longword value to the ASCII representation of the value's hexadecimal equivalent. It requires one parameter: the value to be converted.
IZB	Converts an unsigned byte value to the ASCII representation of the value's decimal equivalent. It requires one parameter: the value to be converted. \$FAO uses only the low-order byte of the longword parameter.
IZW	Converts an unsigned word value to the ASCII representation of the value's decimal equivalent. It requires one parameter: the value to be converted. \$FAO uses only the low-order word of the longword parameter.
IZL	Converts an unsigned longword value to the ASCII representation of the value's decimal equivalent. It requires one parameter: the value to be converted.

Table SYS-2 (Cont.) List of FAO Directives

Directive	Description
Directives for Blank-Filled Numeric Conversion	
IUB	Converts an unsigned byte value to the ASCII representation of the value's decimal equivalent. It requires one parameter: the value to be converted. \$FAO uses only the low-order byte of the longword parameter.
IUW	Converts an unsigned word value to the ASCII representation of the value's decimal equivalent. It requires one parameter: the value to be converted. \$FAO uses only the low-order word of the longword parameter.
IUL	Converts an unsigned longword value to the ASCII representation of the value's decimal equivalent. It requires one parameter: the value to be converted.
ISB	Converts a signed byte value to the ASCII representation of the value's decimal equivalent. It requires one parameter: the value to be converted. \$FAO uses only the low-order byte of the longword parameter.
ISW	Converts a signed word value to the ASCII representation of the value's decimal equivalent. It requires one parameter: the value to be converted. \$FAO uses only the low-order word of the longword parameter.
ISL	Converts a signed longword value to the ASCII representation of the value's decimal equivalent. It requires one parameter: the value to be converted.
Directives for Output String Formatting	
I/	Inserts a new line; that is, a carriage return and line feed. It takes no parameters.
I_	Inserts a tab. It takes no parameters.
I^	Inserts a form feed. It takes no parameters.
!!	Inserts an exclamation point. It takes no parameters.
I%S	Inserts the letter "S" if the most recently converted numeric value is not 1. An uppercase "S" is inserted if the character before the I%S directive is an uppercase character; a lowercase "s" is inserted, if the character is lowercase.
I%T	Inserts the system time. It takes one parameter: the address of a quadword time value to be converted to ASCII. If 0 is specified, the current system time is inserted.
I%U	Converts a longword integer UIC to a standard UIC specification in the format [xxx,yyy], where "xxx" is the group number and "yyy" is the member number. It takes one parameter: a longword integer. The directive inserts the surrounding brackets ([]) and comma (,).
I%I	Converts a longword to the appropriate alphanumeric identifier. If the longword represents a UIC, surrounding brackets ([]) and comma (,) are added as necessary. If no identifier exists and the longword represents a UIC, the longword is formatted as in I%U. Otherwise it is formatted as in IXL with a preceding I%X added to the formatted result.

System Service Descriptions

\$FAO

Table SYS-2 (Cont.) List of FAO Directives

Directive	Description
I%D	Inserts the system date and time. It takes one parameter: the address of a quadword time value to be converted to ASCII. If 0 is specified, the current system date and time is inserted.
In <	See description of next directive (I>).
I>	This directive and the preceding one (In <) are used together to define an output field width of "n" characters within which all data and directives to the right of In< and to the left of I> are left-justified and blank-filled. It takes no parameters.
In*c	Repeats the character "c" in the output string "n" times.
Directives for Parameter Interpretation	
I-	Causes \$FAO to reuse the most recently used parameter in the list. It takes no parameters.
I+	Causes \$FAO to skip the next parameter in the list. It takes no parameters.

Table SYS-3 FAO Output Field Lengths and Fill Characters

Conversion/Substitution Type	Default Length of Output Field	Action When Explicit Output Field Length is Longer than Default	Action When Explicit Output Field Length is Shorter than Default
Hexadecimal			
Byte	2 (zero-filled)	ASCII result is right-justified and blank-filled to the specified length	ASCII result is truncated on the left
Word	4 (zero-filled)		
Longword	8 (zero-filled)		
Octal			
Byte	3 (zero-filled)	Hexadecimal or octal output is always zero-filled to the default output field length then blank-filled to specified length	
Word	6 (zero-filled)		
Longword	11 (zero-filled)		
Signed or Unsigned Decimal	As many characters as necessary	ASCII result is right-justified and blank-filled to the specified length	Signed and unsigned decimal output fields and completely filled with asterisks (_*)
Unsigned Zero-filled Decimal	As many characters as necessary	ASCII result is right-justified and zero-filled to the specified length	
ASCII String Substitution	Length of input character string	ASCII string is left-justified and blank-filled to the specified length	ASCII string is truncated on the right

CONDITION VALUES RETURNED

SS\$_NORMAL	Service successfully completed.
SS\$_BUFFEROVF	Service successfully completed. The formatted output string overflowed the output buffer and has been truncated.
SS\$_ACCVIO	The ctrstr , p1 through pn or prmlst arguments cannot be read, or the outlen argument cannot be written (it may specify 0).
SS\$_BADPARAM	An invalid directive was specified in the FAO control string.

EXAMPLES

Each of the following examples shows an FAO control string with several directives, parameters defined as input for the directives, and the calls to \$FAO to format the output strings. The numbered examples illustrate the following:

- 1 \$FAO macro, !AC, !AS, !AD, and !/ directives
- 2 \$FAO macro, !I, and !AS directives, repeat count, output field length
- 3 \$FAO macro, !UL, !XL, !SL directives
- 4 \$FAOL macro, !UL, !XL, !SL directives
- 5 \$FAOL macro, !UB, !XB, !SB directives
- 6 \$FAO macro, !XW, !ZW, !- directives, repeat count, output field length
- 7 \$FAOL macro, !AS, !UB, !%S, !- directives, variable repeat count
- 8 \$FAO macro, !nc(repeat character), !%D directives
- 9 \$FAO macro, !%D and !%T (with output field lengths), !n (with variable repeat count)
- 10 \$FAO macro, ! < and ! > (define field width), !AC, and !UL directives
- 11 \$FAO macro, !AS and !SL directives

Each example is accompanied by notes. These notes show the output string created by the call to \$FAO and describe in more detail some considerations for using directives. The sample output strings show delta characters (—) in all places where FAO output contains multiple spaces.

System Service Descriptions

\$FAO

Each of the first ten examples refers to the following output fields (these fields are not shown in the data areas for each example):

```
FAODESC:                ; descriptor for output buffer
      .LONG 80           ; output buffer length
      .ADDRESS -
      FAOBUF            ; address of buffer
FAOBUF: .BLKB 80         ; 80-character buffer
FAOLEN: .BLKW 1         ; receive length of output
      .BLKW 1           ; reserve word for $QIO
```

These examples assume that each call to \$FAO will be followed by a call to \$QIO to write the output string produced by \$FAO. The \$QIO system service requires that the length be specified as a longword; therefore, an extra word is provided following the word defined to receive the length of the output string returned by \$FAO.

The final example shows how to make a call to \$FAO from a VAX FORTRAN program.

The examples, numbered one through eleven, follow.

```
1 ; control string and input parameters
;
SLEEPSTR: .ASCID "/SAILORS: !AC !AS !AD" ; descriptor for control
; string
;
WINKEN: .ASCIC /WINKEN/ ; counted ASCII string
BLINKEN:
      .ASCID /BLINKEN/ ; character string descriptor
NOD: .ASCII /NOD/ ; ASCII string
NODLEN: .LONG NODLEN-NOD ; length of ASCII string
;
; call to $FAO
;
      $FAO_S CTRSTR=SLEEPSTR, -
      OUTLEN=FAOLEN, -
      OUTBUF=FAODESC, -
      P1=#WINKEN, -
      P2=#BLINKEN, -
      P3=NODLEN, -
      P4=#NOD
```

\$FAO writes the following output string into FAOBUF:

<CR> <LF> SAILORS: WINKEN BLINKEN NOD

The !/ directive provides a carriage-return/line-feed character (shown as <CR> <LF>) for terminal output.

The !AC directive requires the address of a counted ASCII string (P1 argument); the number sign (#) is required to specify the parameter, so that the PUSHL instruction used by the \$FAO macro pushes the address rather than its contents.

The !AS directive requires the address of a character string descriptor (P2 argument).

The !AD directive requires two parameters: the length of the string to be substituted (P3 argument) and its address (P4 argument).

System Service Descriptions

\$FAO

```
2 ; control string and input parameters
;
NAMESTR:
    .ASCID /UNABLE TO LOCATE !3(8AS)!!/ ; descriptor for
                                           ; control string
;
JONES: .ASCID /JONES/ ; name descriptor
HARRIS: .ASCID /HARRIS/ ; name descriptor
WILSON: .ASCID /WILSON/ ; name descriptor
;
; call to $FAO
;
$FAO_8 CTRSTR=NAMESTR, -
        OUTLEN=FAOLEN, -
        OUTBUF=FAODESC,-
        P1=#JONES, -
        P2=#HARRIS, -
        P3=#WILSON
```

\$FAO writes the following output string into FAOBUF:

UNABLE TO LOCATE JONES____HARRIS____WILSON____!

The !3(8AS) directive contains a repeat count: three parameters (addresses of character string descriptors) are required. \$FAO left-justifies each string into a field of eight characters (the output field length specified).

The double exclamation point directive (!!) supplies a literal exclamation point (!) in the output.

If the directive were specified without an output field length, that is, if the directive were specified as !3(AS), the three output fields would be concatenated, as follows:

UNABLE TO LOCATE JONESHARRISWILSON!

System Service Descriptions

\$FAO

```
3 ; control strings and input parameters for next three examples
;
; descriptor for control string (longword conversion)
LONGSTR:
    .ASCID /VALUES !UL (DEC) !XL (HEX) !SL (SIGNED)/
;
; descriptor for control string (byte conversion)
BYTESTR:
    .ASCID /VALUES !UB (DEC) !XB (HEX) !SB (SIGNED)/
;
VAL1: .LONG 200 ; decimal 200
VAL2: .LONG 300 ; decimal 300
VAL3: .LONG -400 ; negative 400
;
;
; Example 3: Call to $FAO
;
$FAO_S CTRSTR=LONGSTR, -
      OUTBUF=FAODESC, -
      OUTLEN=FAOLEN, -
      P1=VAL1, -
      P2=VAL2, -
      P3=VAL3
```

\$FAO writes the following output string:

VALUES 200 (DEC) 0000012C (HEX) -400 (SIGNED)

The longword value 200 is converted to decimal, the value 300 is converted to hexadecimal, and the value -400 is converted to signed decimal. The ASCII results of each conversion are placed in the appropriate position in the output string.

Note that the hexadecimal output string has eight characters and is zero-filled to the left. This is the default output length for hexadecimal longwords.

```
4 ;
; Call to $FAOL
;
$FAOL_S CTRSTR=LONGSTR, -
      OUTBUF=FAODESC, -
      OUTLEN=FAOLEN, -
      PRMLST=VAL1
```

\$FAO writes the following output string:

VALUES 200 (DEC) 0000012C (HEX) -400 (SIGNED)

The results are the same as the results of Example 3. However, unlike the \$FAO macro, which requires each parameter on the call to be specified, the \$FAOL macro points to a list of consecutive longwords, which \$FAO reads as parameters.

```

5  ;
   ;
   ; Call to $FAOL
   ;
   $FAOL_S CTRSTR=BYTESTR, -
           OUTLEN=FAOLEN, -
           OUTBUF=FAODESC, -
           PRMLST=VAL1

```

Results for Example 5:

\$FAO writes the following output string:

VALUES 200 (DEC) 2C (HEX) 112 (SIGNED)

The input parameters are the same as those for Example 4. However, the control string (BYTESTR) specifies that byte values are to be converted. \$FAO uses the low-order byte of each longword parameter passed to it. The high-order three bytes are not evaluated. Compare these results with the results of Example 4.

```

6  ;
   ; control string
   ;
   MULTSTR:
           .ASCID /HEX: !2(6XW) ZERO-DEC: !2(-)!2(7ZW)/
           .
           .
   ; call to $FAO
   ;
   $FAO_S CTRSTR=MULTSTR, -
           OUTLEN=FAOLEN, -
           OUTBUF=FAODESC, -
           P1=#10000, -
           P2=#9999

```

FAO writes the following output string:

HEX:___2710___270F ZERO-DEC: 00100000009999

Each of the directives !2(6XW) and !2(7ZW) contains repeat counts and output lengths. First, \$FAO performs the !XW directive twice, using the low-order word of the numeric parameters passed. The output length specified is two characters longer than the default output field width of hexadecimal word conversion, so two spaces are placed between the resulting ASCII strings.

The !- directive causes \$FAO to back up over the parameter list. A repeat count is specified with the directive, so that \$FAO skips back over two parameters; then, it uses the same two parameters for the !ZW directive. The !ZW directive causes the output string to be zero-filled to the specified length, in this example, seven characters. Thus, there are no spaces between the output fields.

System Service Descriptions

\$FAO

```

7  ;
; control string and input parameters
;
ARGSTR: .ASCID /!AS RECEIVED !UB ARG!%S: !-!$(4UB)/
;
LISTA: .ADDRESS -
        ORION                ; address of name string
        .LONG 3              ; number of args in list
        .LONG 10             ; argument 1
        .LONG 123            ; argument 2
        .LONG 210            ; argument 3
;
LISTB: .ADDRESS -
        LYRA                 ; address of name string
        .LONG 1              ; number of args in list
        .LONG 255            ; argument 1
;
ORION: .ASCID /ORION/        ; descriptor for process ORION
;
LYRA: .ASCID /LYRA/          ; descriptor for process LYRA
;
; calls to $FAO
;
$FAOL_S CTRSTR=ARGSTR, -
        OUTLEN=FAOLEN, -
        OUTBUF=FAODESC, -
        PRMLST=LISTA
;
$FAOL_S CTRSTR=ARGSTR, -
        OUTLEN=FAOLEN, -
        OUTBUF=FAODESC, -
        PRMLST=LISTB

```

After the first call to \$FAOL shown above, \$FAO writes the following output string:

ORION RECEIVED 3 ARGS:___10 123 210

Following the second call, \$FAO writes the following output string:

LYRA RECEIVED 1 ARG:___255

In each of the examples, the PRMLST argument points to a different parameter list; each list contains, in the first longword, the address of a character string descriptor. The second longword begins an argument list, with the number of arguments remaining in the list. The control string uses this second longword twice: first to output the value contained in the longword, and then to provide the repeat count to output the number of arguments in the list (the !- directive indicates that \$FAO should reuse the parameter).

The !%S directive provides a conditional plural. When the last value converted has a value not equal to 1, \$FAO outputs the character "S"; if the value is a 1 (as in the second example), \$FAO does not output the character "S".

The output field length defines a width of four characters for each byte value converted, to provide spacing between the output fields.

```
8  ;
; control string
;
TIMESTR:
    .ASCID /!5> NOW IS: !%D/
    .
;
; call to $FAO
;
$FAO_S CTRSTR=TIMESTR, -
      OUTLEN=FAOLEN, -
      OUTBUF=FAODESC, -
      P1=#0
```

FAO writes the following output string, where dd-mmm-yyyy is the current day, month, and year, and hh:mm:ss.cc is the current time of day in hours, minutes, seconds, and hundredths of seconds.

> > > > > NOW IS: dd-mmm-yyyy hh:mm:ss.cc

The !5> directive requests \$FAO to write five greater-than (>) characters into the output string. Since there is a space after the directive, \$FAO also writes a space after the greater-than (>) characters on output.

The !%D directive requires the address of a quadword time value, which must be in the system time format. However, when the address of the time value is specified as 0, \$FAO uses the current date and time. For information on how to obtain system time values in the required format, see Section 8, Timer and Time Conversion Services. For a detailed description of the ASCII date and time string returned, see the discussion of the Convert Binary Time to ASCII String (\$ASCTIM) system service in Part II.

```
9  ;
; control string
;
DAYTIMSTR:
    .ASCID /DATE: !11%D!#_TIME: !6%T/
    .
;
; call to $FAO
;
$FAO_S CTRSTR=DAYTIMSTR, -
      OUTLEN=FAOLEN, -
      OUTBUF=FAODESC, -
      P1=#0, -
      P2=#6, -
      P3=#0
```

FAO writes the following output string:

DATE: dd-mmm-yyyy_ _ _ _TIME: hh:mm

In this example, an output length of eleven bytes is specified with the !%D directive, so that \$FAO truncates the time from the date and time string, and outputs only the date.

System Service Descriptions

\$FAO

The !#_ directive requests that the underscore character (_) be repeated the number of times specified by the next parameter. Since P2 is specified as 5, five underscores are written into the output string.

The !%T directive normally returns the full system time. In this example, the !5%T directive provides an output length for the time; only the hours and minutes fields of the time string are written into the output buffer.

```
10 ;
; control string and parameters
;
WIDTHSTR:
.ASCID /!25<VAR: !AC VAL: !UL!>TOTAL: !7UL/
;
VAR1NAME:
.ASCIC /INVENTORY/ ; variable 1 name
VAR1: .LONG 334 ; current value
VAR1TOT: ;
.LONG 6554 ; var 1 total
;
VAR2NAME:
.ASCIC /SALES/ ; var 2 name
VAR2: .LONG 280 ; current value
VAR2TOT: ;
.LONG 10750 ; var 2 total
;
; calls to $FAO
;
$FAO_8 CTRSTR=WIDTHSTR, -
OUTLEN=FAOLEN, -
OUTBUF=FAODESC, -
P1=$VAR1NAME, -
P2=VAR1, -
P3=VAR1TOT
;
$FAO_8 CTRSTR=WIDTHSTR, -
OUTLEN=FAOLEN, -
OUTBUF=FAODESC, -
P1=$VAR2NAME, -
P2=VAR2, -
P3=VAR2TOT
```

After the first call to \$FAO shown above, \$FAO writes the following output string:

VAR: INVENTORY VAL: 334__TOTAL:___6554

After the second call, \$FAO writes the following output string:

VAR: SALES VAL: 280____TOTAL:___10750

The !25 < directive requests an output field width of 25 characters; the end of the field is delimited by the !> directive. Within the field defined in the example above are two directives, !AC and !UL. The strings substituted by these directives can vary in length, but the entire field always has 25 characters.

The !7UL directive formats the longword passed in each example (P2 argument) and right-justifies the result in a 7-character output field.

```

11  INTEGER STATUS,
2     SYS$FAO,
2     SYS$FAOL
! resultant string
CHARACTER*80 OUTSTRING
INTEGER*2    LEN
! array for directives in $FAOL
INTEGER*4    PARAMS(2)
! file name and error number
CHARACTER*80 FILE
INTEGER*4    FILE_LEN,
2            ERROR
! descriptor for $FAOL
INTEGER*4    DESCR(2)
! These variables would generally be set following an error
FILE = '[BOELITZ]TESTING.DAT'
FILE_LEN = 18
ERROR = 25
! call $FAO
STATUS = SYS$FAO ('File !AS aborted at error !SL',
2               LEN,
2               OUTSTRING,
2               FILE(1:FILE_LEN),
2               %VAL(ERROR))
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
TYPE *, 'From SYS$FAO:'
TYPE *, OUTSTRING (1:LEN)
! set up descriptor for filename
DESCR(1) = FILE_LEN ! length
DESCR(2) = %LOC(FILE) ! address
! set up array for directives
PARAMS(1) = %LOC(DESCR) ! file name
PARAMS(2) = ERROR ! error number
! call $FAOL
STATUS = SYS$FAOL ('File !AS aborted at error !SL',
2               LEN,
2               OUTSTRING,
2               PARAMS)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
TYPE *, 'From SYS$FAOL:'
TYPE *, OUTSTRING (1:LEN)
END

```

The above example shows a segment of a VAX FORTRAN program used to output the following string:

FILE [BOELITZ]TESTING.DAT ABORTED AT ERROR 25

System Service Descriptions

\$FILESCAN

\$FILESCAN—Scan String for File Specification

The Scan String for File Specification service searches a string for a file specification and parses the components of that file specification.

FORMAT **SY\$FILESCAN** *srcstr, value1st, [fldflags]*

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS **srcstr**

VMS Usage: **char_string**
type: **character-coded text string**
access: **read only**
mechanism: **by descriptor—fixed length string descriptor**

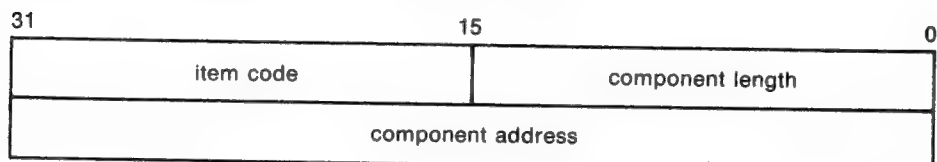
String to be searched for the file specification. The **srcstr** argument is the address of a descriptor pointing to this string.

value1st

VMS Usage: **item_list_2**
type: **longword (unsigned)**
access: **modify**
mechanism: **by reference**

Item list specifying which components of the file specification are to be returned by \$FILESCAN; the components are the node, device, directory, file name, file type and version number. The **itmlst** argument is the address of a list of item descriptors wherein each item descriptor specifies one component. The list of item descriptors is terminated by a longword of 0.

The following diagram depicts a single item descriptor:



ZK-1709-84

\$FILESCAN Item Descriptor Fields

component length

A word in which \$FILESCAN writes the length (in characters) of the requested component. If \$FILESCAN does not locate the component, it returns the value 0 in this field and in the **component address** field and returns the SS\$_NORMAL condition value.

item code

A user-supplied, word-length symbolic code that specifies the component desired. The item codes are defined by the \$FSCNDEF macro. Each item code is described below.

component address

A longword in which \$FILESCAN writes the starting address of the component. This address points to a location in the input string itself.

\$FILESCAN Item Codes

FSCN\$_FILESPEC

When FSCN\$_FILESPEC is specified, \$FILESCAN returns the length and starting address of the full file specification. The full file specification may consist of the node, device, directory, name, type, and version.

FSCN\$_NODE

When FSCN\$_NODE is specified, \$FILESCAN returns the length and starting address of the node name. The node name includes the double colon (::), as well as an access control string (if present).

FSCN\$_DEVICE

When FSCN\$_DEVICE is specified, \$FILESCAN returns the length and starting address of the device name. The device name includes the single colon (:).

FSCN\$_ROOT

When FSCN\$_ROOT is specified, \$FILESCAN returns the length and starting address of the root directory string. The root directory name string includes the opening and closing brackets ([]) or angle brackets (< >).

FSCN\$_DIRECTORY

When FSCN\$_DIRECTORY is specified, \$FILESCAN returns the length and starting address of the directory name. The directory name includes the opening and closing brackets ([]) or angle brackets (< >).

FSCN\$_NAME

When FSCN\$_NAME is specified, \$FILESCAN returns the length and starting address of the file name. The file name does not include any syntactical elements.

In addition, when FSCN\$_NAME is specified, \$FILESCAN will return the length and starting address of a quoted file specification following a node name (as in the specification NODE::"FILE-SPEC"). The beginning and ending quotation marks are included.

FSCN\$_TYPE

When FSCN\$_TYPE is specified, \$FILESCAN returns the length and starting address of the file type. The file type includes the preceding period (.).

System Service Descriptions

\$FILESCAN

FSCN\$_VERSION

When FSCN\$_VERSION is specified, \$FILESCAN returns the length and starting address of the file version number. The file version number includes the preceding period (.) or semicolon (;) delimiter.

fldflags

VMS Usage: **mask_longword**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Longword flag mask in which \$FILESCAN sets a bit for each file specification component found in the input string. The **fldflags** argument is the address of this longword flag mask.

A symbolic name for each significant flag bit is defined by the \$FSCNDEF macro. The following list gives the file specification component which corresponds to the symbolic name of each flag bit.

Symbolic Name	Corresponding Component
FSCN\$V_NODE	Node name
FSCN\$V_DEVICE	Device name
FSCN\$V_ROOT	Root directory name string
FSCN\$V_DIRECTORY	Directory name
FSCN\$V_NAME	File name
FSCN\$V_TYPE	File type
FSCN\$V_VERSION	Version number

The **fldflags** argument is optional. Specify **fldflags** instead of **value1st** when you want to know which components of a file specification are present in a string, but do not need to know the contents or length of these components.

DESCRIPTION

When \$FILESCAN locates a partial file specification (for example, DISK:[FOO]), it returns the length and starting address of those components that were both requested in the item list and that were found in the string. If a component was requested in the item list but not found in the string, \$FILESCAN returns a length of 0 and starting address of 0 to the **component length** and **component address** fields of the item descriptor for that component.

The information returned about all individual components, when taken together, describes the entire contiguous file specification string. For example, to extract only the file name and file type from a full file specification string, you can add the length of these two components and use the address of the first component (file name).

The \$FILESCAN service does not perform comprehensive syntax checking. Specifically, it does not check that a component has a valid length.

However, \$FILESCAN does make the following checks:

- The component must have required syntactical elements; for example, a directory component must be enclosed in brackets and a node name must be followed by a double colon (::).

System Service Descriptions

\$FILESCAN

- The component must not contain invalid characters. Invalid characters are specific to each component. For example, a comma (,) is a valid character in a directory component but not in a file type component.
- Spaces, tabs, and carriage returns are permitted within quoted strings, but are invalid anywhere else.

Invalid characters are treated as terminators. For example, if \$FILESCAN encounters a space within a file name component, it assumes that the space terminates the full file specification string.

The \$FILESCAN service recognizes the DEC Multinational alphabetical characters (such as a') as alphanumeric characters.

The \$FILESCAN service does not (1) assume default values for unspecified file specification components, (2) perform logical name translation on components, (3) perform wild card processing, or (4) perform directory lookups.

CONDITION VALUES RETURNED

SS\$_NORMAL

Successful completion.

SS\$_ACCVIO

The service could not read the string pointed to by the **srcstr** argument or could not write to an item descriptor in the item list specified by the **valuelst** argument.

SS\$_BADPARAM

The item list contains an invalid item code.

System Service Descriptions

\$FIND_HELD

CONDITION VALUES RETURNED

SS\$_NORMAL

Successful completion.

SS\$_ACCVIO

The **id** argument cannot be read by the caller, or the **holder**, **attrib**, or **ctxt** arguments cannot be written by the caller.

SS\$_IVCHAN

The contents of the **ctxt** longword are not valid.

SS\$_INSFMEM

Insufficient process dynamic memory is available to open the rights database.

SS\$_IVIDENT

The specified holder identifier is of invalid format.

SS\$_NOIOCHAN

No more rights database context streams are available.

SS\$_NOSUCHID

The specified holder identifier does not exist, or no further identifiers are held by the specified holder.

RMS\$_PRV

The user does not have read access to the rights database.

Since the rights database is an indexed file accessed with VAX RMS, this service may also return RMS status codes associated with operations on indexed files. Please refer to the *VAX Record Management Services Reference Manual* for descriptions of these status codes.

\$FIND_HOLDER—Find Holder of Identifier

The Find Holder of Identifier service returns the holder of a specified identifier. When called repeatedly with a context longword, it returns all the holders of the specified identifier in the order in which they were added.

FORMAT **SY\$FIND_HOLDER** *id* ,[*holder*] ,[*attrib*] ,[*contxt*]

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED".

ARGUMENTS *id*

VMS Usage: **rights_id**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Binary identifier value whose holder(s) is found by \$FIND_HOLDER. The *id* argument is a longword containing the binary identifier value.

holder

VMS Usage: **rights_holder**
type: **quadword (unsigned)**
access: **write only**
mechanism: **by reference**

Holder identifier returned when \$FIND_HOLDER completes execution. The **holder** argument is the address of a quadword containing the holder identifier. The first longword contains the UIC of the holder with the high-order word containing the group number and the low-order word containing the member number. The second longword contains the value zero.

attrib

VMS Usage: **mask_longword**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Mask of attributes associated with the holder record specified by **holder**. The **attrib** argument is the address of a longword containing the attribute mask.

Symbol values are offsets to the bits within the longword. You can also obtain the values as masks with the appropriate bit set using the prefix KGB\$M rather than KGB\$V. The symbols are defined in the system macro library (\$KGBDEF).

System Service Descriptions

\$FIND_HOLDER

Bit	Meaning When Set
KGB\$V_DYNAMIC	Allows the unprivileged holder to add or remove the identifier from the process rights list
KGB\$V_RESOURCE	Allows the holder to charge resources, such as disk blocks, to the identifier

ctxt

VMS Usage: **context**
type: **longword (unsigned)**
access: **modify**
mechanism: **by reference**

Context value used while searching for all the holders of the specified identifier when executing \$FIND_HOLDER. The **ctxt** argument is the address of a longword containing the context value. When calling \$FIND_HOLDER repeatedly, **ctxt** must be set initially to zero and the resulting context of each call to \$FIND_HOLDER must be presented to each subsequent call. Once the argument has been passed to SYS\$FIND_HOLDER, do not modify its value.

DESCRIPTION

The Find Holder of Identifier service returns the holder of the specified identifier. To determine all the holders of the specified identifier, call SYS\$FIND_HOLDER repeatedly until it returns the status code SS\$_NOSUCHID. SS\$_NOSUCHID indicates that \$FIND_HOLDER has returned all identifiers, cleared the context longword, and deallocated the record stream. If you complete your calls to \$FIND_HOLDER before SS\$_NOSUCHID is returned, use the \$FINISH_RDB service to clear the context value and deallocate the record stream.

Note that when you use wildcarding with this service, the records are returned in the order that they were originally written. (This action results from the fact that the first record is located on the basis of the identifier. Thus, all the target records have the same identifier, or in other words, they have duplicate keys, which leads to retrieval in the order in which they were written.)

CONDITION VALUES RETURNED

SS\$_NORMAL	Successful completion.
SS\$_ACCVIO	The id argument cannot be read by the caller, or the holder, attrib, or ctxt arguments cannot be written by the caller.
SS\$_IVCHAN	The contents of the context longword are not valid.
SS\$_INSFMEM	Insufficient process dynamic memory is available to open the rights database.
SS\$_IVIDENT	The specified identifier or holder identifier is of invalid format.
SS\$_NOIOCHAN	No more rights database context streams are available.

System Service Descriptions

\$FIND_HOLDER

SS\$_NOSUCHID

The specified identifier does not exist in the rights database, or no further holders exist for the specified identifier.

RMS\$_PRV

The user does not have read access to the rights database.

Since the rights database is an indexed file accessed with VAX RMS, this service may also return RMS status codes associated with operations on indexed files. Please refer to the *VAX Record Management Services Reference Manual* for descriptions of these status codes.

System Service Descriptions

\$FINISH_RDB

\$FINISH_RDB—Terminate Rights Database Context

The Terminate Rights Database Context service deallocates the record stream and clears the context value used with \$FIND_HELD, \$FIND_HOLDER, or \$IDTOASC.

FORMAT **SY\$FINISH_RDB** *ctxt*

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS *ctxt*

VMS Usage: **context**
type: **longword (unsigned)**
access: **modify**
mechanism: **by reference**

Context value to be cleared when \$FINISH_RDB completes execution. The *ctxt* argument is a longword containing the address of the context value.

DESCRIPTION

The \$FINISH_RDB service clears the context longword and deallocates the record stream associated with a sequence of rights database lookups performed by the \$IDTOASC, \$FIND_HOLDER, and \$FIND_HELD services.

If you repeatedly call \$IDTOASC, \$FIND_HOLDER, or \$FIND_HELD until SS\$_NOSUCHID is returned, \$FINISH_RDB does not need to be called because the record stream has already been deallocated and the context longword has already been cleared.

**CONDITION
VALUES
RETURNED**

SS\$_NORMAL
SS\$_ACCVIO
SS\$_IVCHAN

Service is successfully completed.

The *ctxt* argument cannot be written by the caller.

The contents of the context longword are not valid.

Since the rights database is an indexed file accessed with VAX RMS, this service may also return RMS status codes associated with operations on indexed files. Please refer to the *VAX Record Management Services Reference Manual* for descriptions of these status codes.

\$FORCEX—Force Exit

The Force Exit system service causes an Exit (\$EXIT) service call to be issued on behalf of a specified process.

FORMAT

SY\$FORCEX [*pidadr*] , [*prcnam*] , [*code*]

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS

pidadr

VMS Usage: **process_id**
type: **longword (unsigned)**
access: **modify**
mechanism: **by reference**

Process identification (PID) of the process to be forced to exit. The **pidadr** argument is the address of a longword containing the PID.

The **pidadr** is optional, but it must be specified if the process that is to be forced to exit is not in the same UIC group as the calling process.

If neither the **pidadr** nor **prcnam** argument is specified, the caller is forced to exit and control is not returned.

prcnam

VMS Usage: **process_name**
type: **character-coded text string**
access: **read only**
mechanism: **by descriptor—fixed length string descriptor**

Process name of the process that is to be forced to exit. The **prcnam** argument is the address of a character string descriptor pointing to a 1- to 15-character process name string.

The **prcnam** argument can only be used on behalf of processes in the same UIC group as the calling process. To force processes in other groups to exit, the **pidadr** argument must be specified. This restriction exists because VAX/VMS interprets the UIC group number of the calling process as part of the specified process name; the names of processes are unique to UIC groups.

If neither the **pidadr** nor **prcnam** argument is specified, the caller is forced to exit and control is not returned.

System Service Descriptions

\$FORCEX

code

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Completion code value to be used as the exit parameter. The **code** argument is a longword containing this value. If **code** is not specified, a value of 0 is passed as the completion code.

DESCRIPTION

Depending on the operation, use of \$FORCE may require the calling process to have certain privilege:

- GROUP privilege is required to force an exit for a process in the same group that does not have the same UIC as the calling process.
- WORLD privilege is required to force an exit for any process in the system

The Force Exit system service requires system dynamic memory.

The image executing in the target process follows normal exit procedures. For example, if any exit handlers have been specified, they gain control before the actual exit occurs. Use the Delete Process (\$DELPRC) service if you do not want a normal exit.

When a forced exit is requested for a process, a user mode AST is queued for the target process. The AST routine causes the \$EXIT service call to be issued by the target process. Because the AST mechanism is used, user mode ASTs must be enabled for the target process, or no exit occurs until ASTs are reenabled. Thus, for example, a suspended process cannot be stopped by \$FORCEX. The process that calls \$FORCEX receives no notification that the exit is not being performed.

The \$FORCEX service completes successfully if a force exit request is already in effect for the target process but the exit is not yet completed.

CONDITION VALUES RETURNED

SS\$_NORMAL	Service successfully completed.
SS\$_ACCVIO	The process name string or string descriptor cannot be read by the caller, or the process identification cannot be written by the caller.
SS\$_IVLOGNAM	The process name string has a length equal to 0 or greater than 15.
SS\$_NONEXPR	Warning. The specified process does not exist, or an invalid process identification was specified.
SS\$_NOPRIV	The process does not have the privilege to force an exit for the specified process.
SS\$_INSFMEM	Insufficient system dynamic memory is available for the operation.

\$FORMAT_ACL—Format Access Control List Entry

The Format Access Control List Entry service formats the specified ACL entry (ACE) into a text string.

FORMAT	SYS\$FORMAT_ACL <i>aclent</i> , <i>[acllen]</i> , <i>aclstr</i> , <i>[width]</i> , <i>[trmdsc]</i> , <i>[indent]</i> , <i>[accnam]</i>
---------------	---

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS *aclent*

VMS Usage: **char_string**
type: **character-coded text string**
access: **read only**
mechanism: **by descriptor—fixed length string descriptor**

Description of the ACE formatted when \$FORMAT_ACL completes execution. The *aclent* argument is the address of a descriptor pointing to a buffer containing the description of the input ACE. The first byte of the buffer contains the length of the ACE; the second byte contains a value that identifies the type of ACE, which in turn determines the ACE format.

See the Description section for more information on ACE format.

acllen

VMS Usage: **word_unsigned**
type: **word (unsigned)**
access: **write only**
mechanism: **by reference**

Length of the output string resulting when \$FORMAT_ACL completes execution. The *acllen* argument is the address of a word containing the number of characters written to *aclstr*.

aclstr

VMS Usage: **char_string**
type: **character-coded text string**
access: **write only**
mechanism: **by descriptor—fixed length string descriptor**

Formatted ACE resulting when \$FORMAT_ACL completes its execution. The *aclstr* argument is the address of a string descriptor pointing to a buffer containing the output string.

System Service Descriptions

\$FORMAT_ACL

width

VMS Usage: **word_unsigned**
type: **word (unsigned)**
access: **read only**
mechanism: **by reference**

Maximum width of the formatted ACE resulting when \$FORMAT_ACL completes its execution. The **width** argument is the address of a word containing the maximum width of the formatted ACE. If this argument is omitted or contains zero, an infinite length display line is assumed. When the width is exceeded, the character specified by **trmdsc** is inserted.

trmdsc

VMS Usage: **char_string**
type: **character-coded text string**
access: **read only**
mechanism: **by descriptor—fixed length string descriptor**

Line termination character(s) used in the formatted ACE. The **trmdsc** argument is the address of a descriptor pointing to character string containing the termination character(s) for each formatted ACE that are inserted when the width has been exceeded.

indent

VMS Usage: **word_unsigned**
type: **word (unsigned)**
access: **read only**
mechanism: **by reference**

Number of blank characters beginning each line of the formatted ACE. The **indent** argument is the address of a word containing the number of blank characters you want inserted at the beginning of each formatted ACE.

accnam

VMS Usage: **access_bit_names**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Names of the bits in the access mask when executing the \$FORMAT_ACL. The **accnam** argument is the address of an array of 32 quadword descriptors that define the names of the bits in the access mask. Each element points to the name of a bit. The first element names bit 0, the second element names bit 1 and so on. If **accnam** is omitted, the following names are used.

Bit 0 **READ**
Bit 1 **WRITE**
Bit 2 **EXECUTE**
Bit 3 **DELETE**
Bit 4 **CONTROL**
Bit 5 **BIT_5**
Bit 6 **BIT_6**

Bit 31 **BIT_31**

System Service Descriptions

\$FORMAT_ACL

DESCRIPTION

The Format Access Control List Entry service formats the specified ACL entry (ACE) into text string representation. The format for ACE type is described in the following sections. The byte offsets and type values are defined in the system macro library (\$ACEDEF).

Alarm Ace

The access alarm ace sets a security alarm. Its format is as follows.

flags	type	length
access		
alarm name		

ZK-1710-84

Field	Symbol Name	Description
length	ACE\$B_SIZE	Byte containing the length in bytes of the ACE buffer
type	ACE\$B_TYPE	Byte containing the type value ACE\$C_ALARM
flags	ACE\$W_FLAGS	Word containing alarm ACE information and ACE type-independent information
access	ACE\$L_ACCESS	Longword containing a mask indicating the access modes to be watched
alarm name	ACE\$T_AUDITNAME	Counted character string containing the alarm name

The flag field contains information specific to alarm ACEs and information applicable to all types of ACEs. The following symbols are bit offsets to the alarm ACE information.

Bit	Meaning When Set
ACE\$V_SUCCESS	Indicates that the alarm is raised when access is successful
ACE\$V_FAILURE	Indicates that the alarm is raised when access fails

The following symbols are bit offsets to ACE information that is independent of ACE type.

Bit	Meaning When Set
ACE\$V_DEFAULT	This ACE is added to the ACL of any file created in the directory whose ACL contains this ACE. This option is applicable only for an ACE in a directory file's ACL.

System Service Descriptions

\$FORMAT_ACL

Bit	Meaning When Set
ACE\$V_HIDDEN	This ACE is application dependent. The DCL ACL commands and the ACL editor cannot be used to change the setting; the DCL command DIRECTORY /ACL does not display it.
ACE\$V_NOPROPAGATE	This ACE is not propagated among versions of the same file.
ACE\$V_PROTECTED	This ACE is not deleted if the entire ACL is deleted; instead this ACE must be explicitly deleted.

The following symbol values are offsets to bits within the access mask. You can also obtain the symbol values as masks with the appropriate bit set using the prefix ACE\$M rather than ACE\$V.

Bit	Meaning When Set
ACE\$V_READ	Read access is monitored.
ACE\$V_WRITE	Write access is monitored.
ACE\$V_EXECUTE	Execute access is monitored.
ACE\$V_DELETE	Delete access is monitored.
ACE\$V_CONTROL	Modification of the access field is monitored.

Application Ace

The application ACE contains application dependent information. Its format is as follows.

flags	type	length
application mask		
. . application information . .		

ZK-1711-84

Field	Symbol Name	Description
length	ACE\$B_SIZE	Byte containing the length in bytes of the ACE buffer.
type	ACE\$B_TYPE	Byte containing the type value ACE\$C_INFO.
flags	ACE\$W_FLAGS	Word containing application ACE information and ACE type-independent information.
application mask	ACE\$L_INFO_FLAGS	Longword containing a mask defined and used by the application.

System Service Descriptions

\$FORMAT_ACL

Field	Symbol Name	Description
application information	ACE\$_INFO_START	Variable length data structure defined and used by the application. The length of this data is implied by length field.

The flag field contains information specific to application ACEs and information applicable to all types of ACEs. The following symbols are bit offsets to the application ACE information.

Bit	Meaning When Set
ACE\$_INFO_TYPE	Four-bit field containing a value indicating whether the application is a CSS application (ACE\$_CSS) or a customer application (ACE\$_CUST)

The following symbols are bit offsets to ACE information that is independent of ACE type.

Bit	Meaning When Set
ACE\$_DEFAULT	This ACE is added to the ACL of any file created in the directory whose ACL contains this ACE. This bit is applicable only for an ACE in a directory file's ACL.
ACE\$_HIDDEN	This bit is application dependent. The DCL ACL commands and the ACL editor cannot be used to change the setting; the DCL command DIRECTORY /ACL does not display it.
ACE\$_NOPROPAGATE	This ACE is not propagated between versions of the same file.
ACE\$_PROTECTED	This ACE is not deleted if the entire ACL is deleted; instead this ACE must be explicitly deleted.

Directory Default Ace

The directory default ACE specifies the UIC-based protection for all files created in the directory. This type of ACE can be used only in the ACL of a directory file. Its format is as follows.

flags	type	length
spare		
system		
owner		
group		
world		

ZK-1712-84

System Service Descriptions

\$FORMAT_ACL

Field	Symbol Name	Description
length	ACE\$B_SIZE	Byte containing the length in bytes of the ACE buffer.
type	ACE\$B_TYPE	Byte containing the type value ACE\$C_DIRDEF.
flags	ACE\$W_FLAGS	Word containing ACE type independent information.
spare	ACE\$L_SPARE1	Longword that is reserved for future use and must be zero.
system	ACE\$L_SYS_PROT	Longword containing a mask indicating the access mode granted to system users. Each bit represents one type of access.
owner	ACE\$L_OWN_PROT	Longword containing a mask indicating the access mode granted to the owner. Each bit represents one type of access.
group	ACE\$L_GRP_PROT	Longword containing a mask indicating the access mode granted to group users. Each bit represents one type of access.
world	ACE\$L_WOR_PROT	Longword containing a mask indicating the access mode granted to the world. Each bit represents one type of access.

The flag field contains information applicable to all types of ACEs. The following symbols are bit offsets to ACE information that is independent of ACE type.

Bit	Meaning When Set
ACE\$V_DEFAULT	This ACE is added to the ACL of any file created in the directory whose ACL contains this ACE. This option is applicable only for an ACE in a directory file's ACL.
ACE\$V_HIDDEN	This ACE is application dependent. The DCL ACL commands and the ACL editor cannot be used to change the setting; the DCL command DIRECTORY /ACL does not display it.
ACE\$V_NOPROPAGATE	This ACE is not propagated among versions of the same file.
ACE\$V_PROTECTED	This ACE is not deleted if the entire ACL is deleted; instead this ACE must be explicitly deleted.

The system interprets the bits within the access mask as shown below. The following symbol values are offsets to bits within the mask indicating the access mode granted in the system, owner, group, and world fields.

Bit	Meaning When Set
ACE\$V_READ	Read access is granted.
ACE\$V_WRITE	Write access is granted.
ACE\$V_EXECUTE	Execute access is granted.
ACE\$V_DELETE	Delete access is granted.

System Service Descriptions

\$FORMAT_ACL

You can also obtain the symbol values as masks with the appropriate bit set by using the prefix ACE\$M rather than ACE\$V.

Identifier Ace

The identifier ACE controls access to an object based on identifiers. Its format is as follows.

flags	type	length
access		
reserved		
reserved		
.		
.		
.		
identifier		
identifier		
.		
.		
.		

ZK-1713-84

Field	Symbol Name	Description
length	ACE\$B_SIZE	Byte containing the length in bytes of the ACE buffer.
type	ACE\$B_TYPE	Byte containing the type value ACE\$C_KEYID.
flags	ACE\$W_FLAGS	Word containing identifier ACE information and ACE type-independent information.
access	ACE\$L_ACCESS	Longword containing a mask indicating the access mode granted to the specified identifiers.
reserved	ACE\$V_RESERVED	Longwords containing application-specific information. The number of reserved longwords is specified in the flags field.

System Service Descriptions

\$FORMAT_ACL

Field	Symbol Name	Description
identifier	ACE\$_KEY	Longwords containing identifiers. The number of longwords is implied by ACE\$_LENGTH. If an accessor holds all of the listed identifiers, the ACE is said to match the accessor and the access specified in ACE\$_ACCESS is granted.

The flag field contains information specific to identifier ACEs and information applicable to all types of ACEs. The following symbol is a bit offset to identifier ACE information.

Bit	Meaning When Set
ACE\$_RESERVED	Four-bit field containing the number of longwords to reserve for application dependent data. The number must be between 0 and 15. The reserved longwords, if any, immediately precede the identifiers.

The following symbols are bit offsets to ACE information that is independent of ACE type.

Bit	Meaning When Set
ACE\$_DEFAULT	This ACE is added to the ACL of any file created in the directory whose ACL contains this ACE. This bit is applicable only for an ACE in a directory file's ACL.
ACE\$_HIDDEN	This bit is application dependent. The DCL ACL commands and the ACL editor cannot be used to change the setting; the DCL command DIRECTORY /ACL does not display it.
ACE\$_NOPROPAGATE	This ACE is not propagated between versions of the same file.
ACE\$_PROTECTED	This ACE is not deleted if the entire ACL is deleted; instead this ACE must be explicitly deleted.

The following symbol values are offsets to bits within the mask indicating the access mode granted in the system, owner, group, and world fields.

Bit	Meaning When Set
ACE\$_READ	Read access is granted.
ACE\$_WRITE	Write access is granted.
ACE\$_EXECUTE	Execute access is granted.
ACE\$_DELETE	Delete access is granted.
ACE\$_CONTROL	Modification of the access field is granted.

You can also obtain the symbol values as masks with the appropriate bit set by using the prefix ACE\$_M rather than ACE\$_V.

System Service Descriptions

\$FORMAT_ACL

CONDITION VALUES RETURNED

SS\$_NORMAL

Service successfully completed.

SS\$_ACCVIO

The ACL entry or its descriptor cannot be read by the caller, or the string descriptor cannot be read by the caller, or the length word or the string buffer cannot be written by the caller.

SS\$_BUFFEROVF

Service successfully completed. The output string has overflowed the buffer and has been truncated.

\$GETDVI—Get Device/Volume Information

The Get Device/Volume Information service returns information about an I/O device; this information consists of primary and secondary device characteristics.

The \$GETDVI service completes asynchronously; that is, it returns to the caller after queuing the information request, without waiting for the requested information to be returned.

For synchronous completion, use the Get Device/Volume Information and Wait (\$GETDVIW) service. The \$GETDVIW service is identical to the \$GETDVI service in every way except that \$GETDVIW returns to the caller with the requested information.

For additional information about system service completion, refer to the Synchronize (\$SYNCH) service and to Section 2.5.

FORMAT	SYSS\$GETDVI <i>[efn] ,[chan] ,[devnam] ,itmlst ,[iosb] ,[astadr] ,[astprm] ,[nullarg]</i>
---------------	---

RETURNS	VMS Usage: cond_value type: longword (unsigned) access: write only mechanism: by value
----------------	---

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS *efn*

VMS Usage: **ef_number**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Number of the event flag to be set when \$GETDVI returns the requested information. The *efn* argument is a longword containing this number.

Upon request initiation, \$GETDVI clears the specified event flag (or event flag 0 if *efn* was not specified). Then, when \$GETDVI returns the requested information, it sets the specified event flag (or event flag 0).

chan

VMS Usage: **channel**
type: **word (unsigned)**
access: **read only**
mechanism: **by value**

Number of the I/O channel assigned to the device about which information is desired. The *chan* argument is a word containing this number.

System Service Descriptions

\$GETDVI

To identify a device to \$GETDVI, you can specify either the **chan** argument or the **devnam** argument, but you should not specify both. If both arguments are specified, the **chan** argument is used.

If neither **chan** nor **devnam** is specified, \$GETDVI uses a default value of 0 for **chan**.

devnam

VMS Usage: **device_name**

type: **character-coded text string**

access: **read only**

mechanism: **by descriptor—fixed length string descriptor**

The name of the device about which \$GETDVI is to return information. The **devnam** argument is the address of a character string descriptor pointing to this name string.

The device name string may be either a physical device name or a logical name. If the first character in the string is an underscore (_), the string is considered a physical device name; otherwise, the string is considered a logical name and logical name translation is performed until either a physical device name is found or the system default number of translations has been performed.

If the device name string contains a colon, the colon and the characters that follow it are ignored.

To identify a device to \$GETDVI, you can specify either the **chan** argument or the **devnam** argument, but you should not specify both. If both arguments are specified, the **chan** argument is used.

If neither **chan** nor **devnam** is specified, \$GETDVI uses a default value of 0 for **chan**.

itmlst

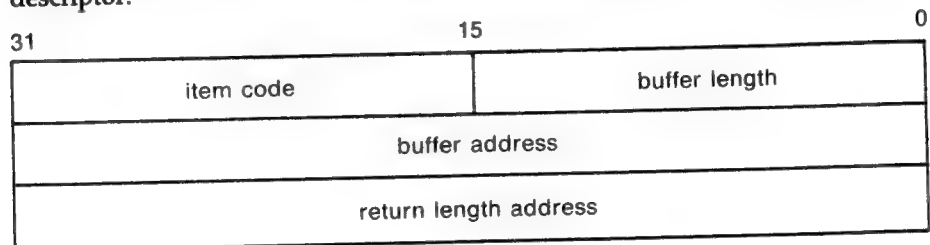
VMS Usage: **item_list_3**

type: **longword (unsigned)**

access: **read only**

mechanism: **by reference**

Item list specifying which information about the device is to be returned. The **itmlst** argument is the address of a list of item descriptors, each of which describes an item of information. The list of item descriptors is terminated by a longword of 0. The following diagram depicts the format of a single item descriptor.



ZK-1705-84

System Service Descriptions

\$GETDVI

\$GETDVI Item Descriptor Fields

buffer length

A word containing a user-supplied integer specifying the length (in bytes) of the buffer in which \$GETDVI is to write the information. The length of the buffer needed depends upon the item code specified in the **item code** field of the item descriptor. If the value of **buffer length** is too small, \$GETDVI truncates the data.

item code

A word containing a user-supplied symbolic code specifying the item of information that \$GETDVI is to return. These codes are defined by the \$DVIDEF macro. Each item code is described in the "\$GETDVI Item Codes" section.

buffer address

A longword containing the user-supplied address of the buffer in which \$GETDVI is to write the information.

return length address

A longword containing the user-supplied address of a word in which \$GETDVI writes the length in bytes of the information it actually returned.

\$GETDVI Item Codes

DVIS__ACPPID

When DVIS__ACPPID is specified, \$GETDVI returns the ACP process ID as a 4-byte hexadecimal number.

DVIS__ACPTYPE

When DVIS__ACPTYPE is specified, \$GETDVI returns the ACP type code as a 4-byte hexadecimal number. The following symbols define each of the ACP type codes that \$GETDVI can return.

Symbol	Description
DVISC__ACP_F11V1	Files-11 Level 1
DVISC__ACP_F11V2	Files-11 Level 2
DVISC__ACP_MTA	Magnetic tape
DVISC__ACP_NET	Networks
DVISC__ACP_REM	Remote I/O

DVIS__ALLDEVNAM

When DVIS__ALLDEVNAM is specified, \$GETDVI returns the allocation-class-device-name, which is a 64-byte hexadecimal string. The allocation-class-device-name uniquely identifies each device that is currently connected to any VAX node in a VAXcluster or to a single node VAX. This item code generates a single unique name for a device even if the device is dual-ported.

One use for the allocation-class-device-name might be in an application wherein processes need to coordinate their access to devices (not volumes) using the VAX/VMS lock manager. In this case, the program would make the device a resource to be locked by the VAX/VMS lock manager, specifying as the resource name the following concatenated components: (1) a user facility prefix followed by an underscore character and (2) the allocation-class-device-name of the device.

System Service Descriptions

\$GETDVI

Note that the name returned by the DVI\$_DEVLOCKNAM item code should be used to coordinate access to volumes.

DVI\$_ALLOCLASS

When DVI\$_ALLOCLASS is specified, \$GETDVI returns the allocation class of the host as a longword integer between 0 and 255. An allocation class is a unique number between 0 and 255 that the system manager assigns to a pair of hosts and the dual-path devices which the hosts make available to other nodes in the VAXcluster.

The allocation class provides a way for users to access dual-path devices through either of the hosts that serve them to the VAXcluster. In this way, if one host of an allocation class set is not available, the user can gain access to a device specified by that allocation class through the other host of the allocation class. The user does not have to be concerned with which host of the allocation class provides access to the device. Specifically, the device name string is constructed using the following format.

\$allocation_class\$device_name

To access a device through either of its host nodes, use the format above instead of the format which follows.

node_name\$device_name

For a detailed discussion of allocation classes refer to the *Guide to VAXclusters*.

DVI\$_ALT_HOST_AVAIL

When DVI\$_ALT_HOST_AVAIL is specified, \$GETDVI returns a longword, which is interpreted as Boolean. A value of 1 indicates that the host serving the alternate path is available; a value of 0 indicates that it is not.

The host is the node which makes the device available to other nodes in the VAXcluster. A host node can be either a VAX with an MSCP server or an HSC-50.

A dual-path device is a device that is made available to the VAXcluster by two hosts. Each of the hosts provides access (serves a path) to the device for users. One host serves the primary path; the other host serves the alternate path. The primary path is simply the path which the system creates through the first available host.

The user is not concerned with which host provides access to the device. When accessing a device, the user specifies the allocation class of the desired device, not the name of the host which serves it.

If the host serving the primary path should fail, the system automatically creates a path to the device through the alternate host.

DVI\$_ALT_HOST_NAME

When DVI\$_ALT_HOST_NAME is specified, \$GETDVI returns the name of the host serving the alternate path as a 64-byte zero-filled string.

Refer to the description of the DVI\$_ALT_HOST_AVAIL item code for more information about hosts, dual-path devices and primary and alternate paths.

DVI\$_ALT_HOST_TYPE

When DVI\$_ALT_HOST_TYPE is specified, \$GETDVI returns, as a 4-byte string, the hardware type of the host serving the alternate path. Each hardware type has a symbolic name. The following list gives each symbolic name and the host it denotes.

System Service Descriptions

\$GETDVI

Name	Host
V750	VAX-11/750
V780	VAX-11/780, VAX-11/782
V785	VAX-11/785
HS50	HSC-50

Refer to the description of the DVI\$_ALT_HOST_AVAIL item code for more information about hosts, dual-pathed devices and primary and alternate paths.

DVI\$_CLUSTER

When DVI\$_CLUSTER is specified, \$GETDVI returns the volume cluster size as a 4-byte decimal number. This item code is applicable only to disks.

DVI\$_CYLINDERS

When DVI\$_CYLINDERS is specified, \$GETDVI returns the number of cylinders on the volume as a 4-byte decimal number. This item code is applicable only to disks.

DVI\$_DEVBUFSIZ

When DVI\$_DEVBUFSIZ is specified, \$GETDVI returns the device buffer size (for example, the width of a terminal or the block size of a tape) as a 4-byte decimal number.

DVI\$_DEVCHAR

When DVI\$_DEVCHAR is specified, \$GETDVI returns device-independent characteristics as a 4-byte bit vector. Each characteristic is represented by a bit. When \$GETDVI sets a bit, the device has the corresponding characteristic. Each bit in the vector has a symbolic name. These symbolic names are defined by the \$DEVDEF macro and are listed below.

Symbol	Description
DEV\$_REC	Device is record oriented.
DEV\$_CCL	Device is a carriage control device.
DEV\$_TRM	Device is a terminal.
DEV\$_DIR	Device is directory structured.
DEV\$_SDI	Device is single-directory structured.
DEV\$_SQD	Device is sequential and block-oriented.
DEV\$_SPL	Device is being spooled.
DEV\$_OPR	Device is an operator.
DEV\$_RCT	Disk contains RCT; DEC standard 166 disk.
DEV\$_NET	Device is a network device.
DEV\$_FOD	Device is files oriented.
DEV\$_DUA	Device is dual ported.
DEV\$_SHR	Device is shareable.
DEV\$_GEN	Device is a generic device.
DEV\$_AVL	Device is available for use.
DEV\$_MNT	Device is mounted.

System Service Descriptions

\$GETDVI

Symbol	Description
DEV\$_MBX	Device is a mailbox.
DEV\$_DMT	Device is marked for dismount.
DEV\$_ELG	Device has error logging enabled.
DEV\$_ALL	Device is allocated.
DEV\$_FOR	Device is mounted foreign.
DEV\$_SWL	Device is software write locked.
DEV\$_IDV	Device can provide input.
DEV\$_ODV	Device can provide output.
DEV\$_RND	Device allows random access.
DEV\$_RTM	Device is a real-time device.
DEV\$_RCK	Device has read-checking enabled.
DEV\$_WCK	Device has write-checking enabled.

Note that each device characteristic listed above has its own individual \$GETDVI item code with the format DVI\$_xxxx, where "xxxx" are the characters following the underscore character in the symbolic name for that device characteristic, as shown in the above list.

For example, when the item code DVI\$_REC is specified, \$GETDVI returns a longword value that is interpreted as Boolean. If the value is 0, the device is not record-oriented; if the value is 1, it is record-oriented. This information is identical to that returned in the DEV\$_REC bit of the longword vector specified by the DVI\$_DEVCHAR item code.

All of these device-characteristic item codes require that the buffer specify a longword.

DVI\$_DEVCHAR2

When DVI\$_DEVCHAR2 is specified, \$GETDVI returns additional device-independent characteristics as a 4-byte bit vector. Each bit in the vector, when set, corresponds to a symbolic name. These symbolic names are defined by the \$DEVDEF macro.

DVI\$_DEVCLASS

When DVI\$_DEVCLASS is specified, \$GETDVI returns the device class as a 4-byte decimal number. Each class has a corresponding symbol. These symbols are defined by the \$DCDEF macro. The following list describes each device class symbol.

Symbol	Description
DC\$_DISK	Disk device
DC\$_TAPE	Tape device
DC\$_SCOM	Synchronous communications device
DC\$_CARD	Card reader
DC\$_TERM	Terminal
DC\$_LP	Line printer
DC\$_REALTIME	Real-time
DC\$_MAILBOX	Mailbox
DC\$_MISC	Miscellaneous device

System Service Descriptions

\$GETDVI

DVI\$_DEVDEPEND

When DVI\$_DEVDEPEND is specified, \$GETDVI returns device-dependent characteristics as a 4-byte bit vector. Refer to the *VAX/VMS I/O Reference Volume* to determine what information is returned for a particular device.

Note that, for terminals only, individual \$GETDVI item codes are provided for most of the informational items returned in the DVI\$_DEVDEPEND longword bit vector. The names of these item codes have the format DVI\$_TT_XXXX, where "XXXX" is the characteristic name. The same characteristic name follows the underscore character in the symbolic name for each bit (defined by the \$TTDEF macro) in the DVI\$_DEVDEPEND longword. For example, the DVI\$_TT_NOECHO item code returns the same information as that returned in the DVI\$_DEVDEPEND bit whose symbolic name is TT\$V_NOECHO.

Each such item code requires that the buffer specify a longword value, which is interpreted as Boolean. A value of 0 indicates that the terminal does not have that characteristic; a value of 1 indicates that it does.

The list of these terminal-specific item codes appears at the end of the list of item codes.

DVI\$_DEVDEPEND2

When DVI\$_DEVDEPEND2 is specified, \$GETDVI returns additional device-dependent characteristics as a 4-byte bit vector. Refer to the *VAX/VMS I/O Reference Volume* to determine what information is returned for a particular device.

Note that, for terminals only, individual \$GETDVI item codes are provided for most of the informational items returned in the DVI\$_DEVDEPEND2 longword bit vector. As described for DVI\$_DEVDEPEND, the same characteristic name appears in the item code as appears in the symbolic name defined for each bit in the DVI\$_DEVDEPEND2 longword, except that in the case of DVI\$_DEVDEPEND2 the symbolic names for bits are defined by the \$TT2DEF macro.

The list of these terminal-specific item codes appears at the end of the list of item codes.

DVI\$_DEVLOCKNAM

When DVI\$_DEVLOCKNAM is specified, \$GETDVI returns the device lock name, which is a 64-byte hexadecimal string. The device lock name uniquely identifies each volume or volume set in a VAXcluster or in a single node VAX. This item code is applicable only to disks.

The item code is applicable to all disk volumes and volume sets: mounted, not mounted, mounted shared, mounted private, or mounted foreign.

The device lock name is assigned to a volume when it is first mounted, and this name cannot be changed, even if the volume name itself is changed. This allows any process on any VAX node in a VAXcluster to access a uniquely identified volume.

One use for the device lock name might be in an application wherein processes need to coordinate their access to files using the VAX/VMS lock manager. In this case, the program would make the file a resource to be locked by the VAX/VMS lock manager, specifying as the resource name the following concatenated components: (1) a user facility prefix followed by an underscore character, (2) the device lock name of the volume on which the file resides, and (3) the file id of the file.

DVI\$_DEVNAM

When DVI\$_DEVNAM is specified, \$GETDVI returns the device name as a 64-byte, zero-filled string. The node name is not returned.

DVI\$_DEVSTS

When DVI\$_DEVSTS is specified, \$GETDVI returns device-dependent status information as a 4-byte bit vector. Symbols for the status bits are defined by the \$UCBDEF macro. Refer to the *VAX/VMS I/O Reference Volume* for this device-dependent information.

DVI\$_DEVTYPE

When DVI\$_DEVTYPE is specified, \$GETDVI returns the device type as a 4-byte decimal number. Symbols for the device types are defined by the \$DCDEF macro.

DVI\$_ERRCNT

When DVI\$_ERRCNT is specified, \$GETDVI returns the device's error count as a 4-byte decimal number.

DVI\$_FREEBLOCKS

When DVI\$_FREEBLOCKS is specified, \$GETDVI returns the number of free blocks on a disk as a 4-byte decimal number. This item code is applicable only to disks.

DVI\$_FULLDEVNAM

When DVI\$_FULLDEVNAM is specified, \$GETDVI returns the node name and device name as a 64-byte, zero-filled string.

DVI\$_FULLDEVNAM is useful in a VAXcluster environment because, unlike DVI\$_DEVNAM, DVI\$_FULLDEVNAM returns the name of the VAX node on which the device resides.

One use for the DVI\$_FULLDEVNAM item code might be to retrieve the name of a device in order to have that name displayed on a terminal. However, this name should not be used as a resource name as input to the lock manager; use the name returned by the DVI\$_DEVLOCKNAM item code for locking volumes and the name returned by DVI\$_ALLDEVNAM for locking devices.

DVI\$_HOST_AVAIL

When DVI\$_HOST_AVAIL is specified, \$GETDVI returns a longword, which is interpreted as Boolean. A value of 1 indicates that the host serving the primary path is available; a value of 0 indicates that it is not available.

Refer to the description of the DVI\$_ALT_HOST_AVAIL item code for more information about hosts, dual-pathed devices and primary and alternate paths.

DVI\$_HOST_COUNT

When DVI\$_HOST_COUNT is specified, \$GETDVI returns, as a longword integer, the number of hosts that make the device available to other nodes in the VAXcluster. One or two hosts, but no more, can make a device available to other nodes in the VAXcluster.

Refer to the description of the DVI\$_ALT_HOST_AVAIL item code for more information about hosts, dual-pathed devices and primary and alternate paths.

System Service Descriptions

\$GETDVI

DVI\$_HOST_NAME

When DVI\$_HOST_NAME is specified, \$GETDVI returns (as a 64-byte, zero-filled string) the name of the host serving the primary path.

Refer to the description of the DVI\$_ALT_HOST_AVAIL item code for more information about hosts, dual-pathed devices, and primary and alternate paths.

DVI\$_HOST_TYPE

When DVI\$_HOST_TYPE is specified, \$GETDVI returns, as a 4-byte string, the type of host serving the primary path. Each hardware type has a symbolic name. The following list gives each symbolic name and the host it denotes:

Name	Host
V750	VAX-11/750
V780	VAX-11/780, VAX-11/782
V785	VAX-11/785
HS50	HSC-50

Refer to the description of the DVI\$_ALT_HOST_AVAIL item code for more information about hosts, dual-pathed devices and primary and alternate paths.

DVI\$_LOCKID

When DVI\$_LOCKID is specified, \$GETDVI returns the lock id of the lock on a disk. The VAX/VMS lock manager locks a disk if it is available to all VAX nodes in a VAXcluster and it is either allocated or mounted. A disk is available to all VAX nodes in a VAXcluster if, for example, it is served by an HSC controller or MSCP server, or if it is a dual-ported MASSBUS disk.

The buffer must specify a longword into which \$GETDVI will return the 4-byte hexadecimal lock id.

DVI\$_LOGVOLNAM

When DVI\$_LOGVOLNAM is specified, \$GETDVI returns the logical name of the volume or volume set as a 64-byte string.

DVI\$_MAXBLOCK

When DVI\$_MAXBLOCK is specified, \$GETDVI returns the maximum number of blocks on the volume as a 4-byte decimal number. This item code is applicable only to disks.

DVI\$_MAXFILES

When DVI\$_MAXFILES is specified, \$GETDVI returns the maximum number of files on the volume as a 4-byte decimal number. This item code is applicable only to disks.

DVI\$_MEDIA_ID

When DVI\$_MEDIA_ID is specified, \$GETDVI returns the nondecoded media id as a longword. This item code is applicable only to disks and tapes.

DVI\$_MEDIA_NAME

When DVI\$_MEDIA_NAME is specified, \$GETDVI returns the name of the volume type (for example, RK07 or TA78) as a 64-byte, zero-filled string. This item code is applicable only to disks and tapes.

System Service Descriptions

\$GETDVI

DVI\$_MEDIA_TYPE

When DVI\$_MEDIA_TYPE is specified, \$GETDVI returns the device name prefix of the volume (for example, DM for an RK07 device or MU for a TA78 device) as a 64-byte, zero-filled string. This item code is applicable only to disks and tapes.

DVI\$_MOUNTCNT

When DVI\$_MOUNTCNT is specified, \$GETDVI returns the mount count for the volume as a 4-byte decimal number.

DVI\$_NEXTDEVNAM

When DVI\$_NEXTDEVNAM is specified, \$GETDVI returns the device name of the next volume in the volume set as a 64-byte, zero-filled string. This item code is applicable only to disks.

DVI\$_OPCNT

When DVI\$_OPCNT is specified, \$GETDVI returns the operation count for the volume as a 4-byte decimal number.

DVI\$_OWNUIC

When DVI\$_OWNUIC is specified, \$GETDVI returns the user identification code (UIC) of the owner of the device as a standard 4-byte VAX/VMS UIC.

DVI\$_PID

When DVI\$_PID is specified, \$GETDVI returns the process identification (PID) of the owner of the device as a 4-byte hexadecimal number.

DVI\$_RECSIZ

When DVI\$_RECSIZ is specified, \$GETDVI returns the blocked record size as a 4-byte decimal number.

DVI\$_REFCNT

When DVI\$_REFCNT is specified, \$GETDVI returns the number of channels assigned to the device as a 4-byte decimal number.

DVI\$_REMOTE_DEVICE

When DVI\$_REMOTE_DEVICE is specified, \$GETDVI returns a longword, which is interpreted as Boolean. A value of 1 indicates that the device is a remote device; a value of 0 indicates that it is not a remote device. A remote device is a device which is not directly connected to the local node, but instead is visible through the VAXcluster.

DVI\$_ROOTDEVNAM

When DVI\$_ROOTDEVNAM is specified, \$GETDVI returns the device name of the root volume in the volume set as a 64-byte, zero-filled string. This item code is applicable only to disks.

DVI\$_SECTORS

When DVI\$_SECTORS is specified, \$GETDVI returns the number of sectors per track as a 4-byte decimal number. This item code is applicable only to disks.

DVI\$_SERIALNUM

When DVI\$_SERIALNUM is specified, \$GETDVI returns serial number of the volume as a 4-byte decimal number. This item code is applicable only to disks.

System Service Descriptions

\$GETDVI

DVIS\$_SERVED_DEVICE

When DVIS\$_SERVED_DEVICE is specified, \$GETDVI returns a longword, which is interpreted as Boolean. A value of 1 indicates that the device is a served device; a value of 0 indicates that it is not a served device. A served device is one whose local node makes it available to other nodes in the VAXcluster.

DVIS\$_SHDW_CATCHUP_COPYING

This item code applies only to the volume shadowing option. See the *VAX/VMS Volume Shadowing Manual*.

DVIS\$_SHDW_FAILED_MEMBER

This item code applies only to the volume shadowing option. See the *VAX/VMS Volume Shadowing Manual*.

DVIS\$_SHDW_MASTER

This item code applies only to the volume shadowing option. See the *VAX/VMS Volume Shadowing Manual*.

DVIS\$_SHDW_MASTER_NAME

This item code applies only to the volume shadowing option. See the *VAX/VMS Volume Shadowing Manual*.

DVIS\$_SHDW_MEMBER

This item code applies only to the volume shadowing option. See the *VAX/VMS Volume Shadowing Manual*.

DVIS\$_SHDW_MERGE_COPYING

This item code applies only to the volume shadowing option. See the *VAX/VMS Volume Shadowing Manual*.

DVIS\$_SHDW_NEXT_MBR_NAME

This item code applies only to the volume shadowing option. See the *VAX/VMS Volume Shadowing Manual*.

DVIS\$_STS

When DVIS\$_STS is specified, \$GETDVI returns the device unit status as a 4-byte bit vector. Each bit in the vector, when set, corresponds to a symbolic name that is defined by the \$UCBDEF macro. The following list describes each name:

Symbol	Description
UCB\$V_TIM	Time out enabled.
UCB\$V_INT	Interrupt expected.
UCB\$V_ERLOGIP	Error log in progress on unit.
UCB\$V_CANCEL	Cancel I/O on unit.
UCB\$V_ONLINE	Unit online.
UCB\$V_POWER	Power failed while unit busy.
UCB\$V_TIMEOUT	Unit timed out.
UCB\$V_INTTYPE	Receiver interrupt.
UCB\$V_BSY	Unit is busy.

System Service Descriptions

\$GETDVI

Symbol	Description
UCB\$V_MOUNTING	Device is being mounted.
UCB\$V_DEADMO	Deallocate at dismount.
UCB\$V_VALID	Volume is software valid.
UCB\$V_UNLOAD	Unload volume at dismount.
UCB\$V_TEMPLATE	This UCB is a template UCB from which other UCBs for this device type are made.
UCB\$V_MNTVERIP	Mount verification in progress.
UCB\$V_WRONGVOL	Wrong volume detected during mount verification.
UCB\$V_DELETEUCB	Delete this UCB when reference count equals 0.

DVI\$_TRACKS

When DVI\$_TRACKS is specified, \$GETDVI returns the number of tracks per cylinder as a 4-byte decimal number. This item code is applicable only to disks.

DVI\$_TRANSCNT

When DVI\$_TRANSCNT is specified, \$GETDVI returns the transaction count for the volume as a 4-byte decimal number.

DVI\$_TT_PHYDEVNAM

When DVI\$_TT_PHYDEVNAM is specified, \$GETDVI returns the physical device name associated with a channel number or virtual terminal name. If the caller specifies a device other than a virtual terminal, \$GETDVI returns a null string. \$GETDVI returns the physical device name as a 64-byte zero-filled string.

DVI\$_UNIT

When DVI\$_UNIT is specified, \$GETDVI returns the unit number as a 4-byte decimal number.

DVI\$_VOLCOUNT

When DVI\$_VOLCOUNT is specified, \$GETDVI returns the number of volumes in the volume set as a 4-byte decimal number. This item code is applicable only to disks.

DVI\$_VOLNAM

When DVI\$_VOLNAM is specified, \$GETDVI returns the volume name as a 12-byte zero-filled string.

DVI\$_VOLNUMBER

When DVI\$_VOLNUMBER is specified, \$GETDVI returns the volume number of this volume in the volume set as a 4-byte decimal number. This item code is applicable only to disks.

DVI\$_VOLSETMEM

When DVI\$_VOLSETMEM is specified, \$GETDVI returns a longword value, which is interpreted as Boolean. A value of 1 indicates that the device is part of a volume set; a value of 0 indicates that it is not. This item code is applicable only to disks.

DVI\$_VPROT

When DVI\$_VPROT is specified, \$GETDVI returns the volume protection mask as a standard 4-byte protection mask.

System Service Descriptions

\$GETDVI

DVI\$_TT_XXXX

DVI\$_TT_XXXX is the format for a series of item codes that return information about terminals. This information consists of terminal characteristics. The "XXXX" portion of the item code name specifies a single terminal characteristic.

Each of these item codes requires that the buffer specify a longword into which \$GETDVI will write a 0 or 1: 0 if the terminal does not have the specified characteristic, and 1 if the terminal does have it. The one exception is the DVI\$_TT_PAGE item code, which when specified causes \$GETDVI to return a decimal longword value that is the page size of the terminal.

This terminal-specific information is also available by using the DVI\$_DEVDEPEND and DVI\$_DEVDEPEND2 item codes. Each of these two item codes specify a longword bit vector wherein each bit corresponds to a terminal characteristic; \$GETDVI sets the corresponding bit for each characteristic possessed by the terminal.

Following is a list of the item codes that return information about terminal characteristics. Refer to the description of the F\$GETDVI lexical function in the *VAX/VMS DCL Dictionary* for information about these characteristics.

Terminal-Specific \$GETDVI Item Codes

DVI\$_TT_NOECHO	DVI\$_TT_NOTYPEAHD
DVI\$_TT_HOSTSYNC	DVI\$_TT_TTSYNC
DVI\$_TT_ESCAPE	DVI\$_TT_LOWER
DVI\$_TT_MECHTAB	DVI\$_TT_WRAP
DVI\$_TT_LFFILL	DVI\$_TT_SCOPE
DVI\$_TT_CRFILL	DVI\$_TT_SETSPEED
DVI\$_TT_EIGHTBIT	DVI\$_TT_MBXDSABL
DVI\$_TT_READSYNC	DVI\$_TT_MECHFORM
DVI\$_TT_NOBRDCST	DVI\$_TT_HALFDUP
DVI\$_TT_MODEM	DVI\$_TT_OPER
DVI\$_TT_LOCALECHO	DVI\$_TT_AUTOBAUD
DVI\$_TT_PAGE	DVI\$_TT_HANGUP
DVI\$_TT_MODHANGUP	DVI\$_TT_BRDCSTMBX
DVI\$_TT_DMA	DVI\$_TT_ALTYPEAHD
DVI\$_TT_ANSICRT	DVI\$_TT_REGIS
DVI\$_TT_AVO	DVI\$_TT_EDIT
DVI\$_TT_BLOCK	DVI\$_TT_DECCRT
DVI\$_TT_EDITING	DVI\$_TT_INSERT
DVI\$_TT_DIALUP	DVI\$_TT_SECURE
DVI\$_TT_FALLBACK	DVI\$_TT_DISCONNECT
DVI\$_TT_PASTHRU	DVI\$_TT_SIXEL
DVI\$_TT_PRINTER	DVI\$_TT_APP_KEYPAD
DVI\$_TT_DRCS	DVI\$_TT_SYSPWD
DVI\$_TT_DECCRT	

DVI\$_yyyy

DVI\$_yyyy is the format for a series of item codes that return device-independent characteristics of a device. There is an item code for each device characteristic returned in the longword bit vector specified by the DVI\$_DEVCHAR item code.

In the description of the DVI\$_DEVCHAR item code, there is a list of symbol names, in which each symbol represents a device characteristic. To construct the \$GETDVI item code for each device characteristic, substitute for "yyyy" that portion of the symbol name that follows the underscore character. For example, the DVI\$_REC item code returns the same information as the DEV\$_REC bit in the DVI\$_DEVCHAR longword bit vector.

The buffer for each of these item codes must specify a longword value, which is interpreted as Boolean. \$GETDVI writes the value 1 into the longword if the device has the specified characteristic and the value 0 if it does not.

iosb

VMS Usage: **io_status_block**
type: **quadword (unsigned)**
access: **write only**
mechanism: **by reference**

I/O status block which is to receive the final completion status. The **iosb** is the address of the quadword I/O status block.

When the **iosb** argument is specified, \$GETDVI sets the quadword to zero upon request initiation. Upon request completion, a condition value is returned to the first longword; the second longword is reserved to DIGITAL.

Though this argument is optional, DIGITAL strongly recommends that you specify it for the following reasons.

- If you are using an event flag to signal the completion of the service, you can test the I/O status block for a condition value to be sure that the event flag was not set by an event other than service completion.
- If you are using the \$SYNCH service to synchronize completion of the service, the I/O status block is a required argument for \$SYNCH.
- The condition value returned in R0 and the condition value returned in the I/O status block provide information about different aspects of the call to the \$GETDVI service. The condition value returned in R0 gives you information about the success or failure of the service call itself; the condition value returned in the I/O status block gives you information about the success or failure of the service operation. Therefore, to accurately assess the success or failure of the call to \$GETDVI, you must check the condition values returned in both R0 and the I/O status block.

Refer to Section 2.5 for more information about system service completion.

astadr

VMS Usage: **ast_procedure**
type: **procedure entry mask**
access: **call without stack unwinding**
mechanism: **by reference**

AST service routine to be executed when \$GETDVI completes. The **astadr** is the address of the entry mask of this routine.

System Service Descriptions

\$GETDVI

If **astadr** is specified, the AST routine will execute at the same access mode as the caller of the \$GETDVI service.

astprm

VMS Usage: **user_arg**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

AST parameter to be passed to the AST service routine specified by the **astadr** argument. The **astprm** argument is the longword parameter.

nullarg

VMS Usage: **null_arg**
type: **quadword (unsigned)**
access: **read only**
mechanism: **by reference**

Place-holding argument. This argument is reserved to DIGITAL.

DESCRIPTION

The **chan** argument can be used only if: (1) the channel has already been assigned; and (2) the caller's access mode is equal to or more privileged than the access mode from which the original channel assignment was made.

The caller of \$GETDVI does not need to have a channel assigned to the device about which information is desired.

The \$GETDVI service returns information about both primary device characteristics and secondary device characteristics. By default, \$GETDVI returns information about the primary device characteristics only.

To obtain information about secondary device characteristics, the item code specifying the information desired must be ORed with the code DVI\$C_SECONDARY.

Information about primary and secondary devices may be obtained in a single call to \$GETDVI.

In most cases, the two sets of characteristics (primary and secondary) returned by \$GETDVI are identical. However, the two sets provide different information in the following cases:

- If the device has an associated mailbox, the primary characteristics are those of the assigned device and the secondary characteristics are those of the associated mailbox.
- If the device is a spooled device, the primary characteristics are those of the intermediate device (such as the disk) and the secondary characteristics are those of the spooled device (such as the printer).
- If the device represents a logical link on the network, the secondary characteristics contain information about the link.

Unless otherwise stated in the "item code" description, \$GETDVI only returns information about the local node.

System Service Descriptions

\$GETDVI

CONDITION VALUES RETURNED

SS\$_NORMAL
SS\$_ACCVIO

Service successfully completed.

The device name string descriptor, device name string, or **itmlst** argument cannot be read; or the buffer or return length longword cannot be written by the caller.

SS\$_BADPARAM

The item list contains an invalid item code, or the **return length address** field in an item descriptor specifies less than four bytes for the return length information.

SS\$_EXASTLM
SS\$_IVCHAN

The process has exceeded its AST limit quota.

An invalid channel number was specified, that is, a channel number larger than the number of channels.

SS\$_IVDEVNAM

The device name string contains invalid characters, or neither the **devnam** nor **chan** arguments were specified.

SS\$_IVLOGNAM

The device name string has a length of 0 or has more than 63 characters.

SS\$_NONLOCAL
SS\$_NOPRIV

Warning. The device is on a remote system.

The specified channel is not assigned or was assigned from a more privileged access mode.

SS\$_NOSUCHDEV

Warning. The specified device does not exist on the host system.

CONDITION VALUES RETURNED IN THE I/O STATUS BLOCK

Same as those returned in R0.

\$GETDVIW—Get Device/Volume Information and Wait

The Get Device/Volume Information and Wait service returns information about an I/O device; this information consists of primary and secondary device characteristics.

The \$GETDVIW service completes synchronously; that is, it returns to the caller with the requested information.

For asynchronous completion, use the Get Device/Volume Information (\$GETDVI) service; \$GETDVI returns to the caller after queuing the information request, without waiting for the information to be returned.

In all other respects, \$GETDVIW is identical to \$GETDVI. Refer to the documentation of \$GETDVI for all other information about the \$GETDVIW service.

For additional information about system service completion, refer to the Synchronize (\$SYNCH) service and to Section 2.5.

FORMAT

SYSS\$GETDVIW *[efn] , [chan] , [devnam] , itmlst [, iosb]
[, astadr] [, astprm] [, nullarg]*

\$GETJPI—Get Job/Process Information

The Get Job/Process Information service returns information about one or more processes on the system.

The \$GETJPI service completes asynchronously; that is, it returns to the caller after queuing the information request, without waiting for the requested information to be returned.

For synchronous completion, use the Get Job/Process Information and Wait (\$GETJPIW) service. The \$GETJPIW service is identical to the \$GETJPI service in every way except that \$GETJPIW returns to the caller with the requested information.

For additional information about system service completion, refer to the Synchronize (\$SYNCH) service and to Section 2.5.

FORMAT **SYS\$GETJPI** *[efn] , [pidadr] , [prcnam] , itmlst [, iosb] [, astadr] [, astprm]*

RETURNS

VMS Usage: **cond_value**
 type: **longword (unsigned)**
 access: **write only**
 mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS *efn*

VMS Usage: **ef_number**
 type: **longword (unsigned)**
 access: **read only**
 mechanism: **by value**

Number of the event flag to be set when \$GETJPI returns the requested information. The *efn* argument is a longword containing this number.

Upon request initiation, \$GETJPI clears the specified event flag (or event flag 0 if *efn* was not specified). Then when \$GETJPI returns the requested information, it sets the specified event flag (or event flag 0).

pidadr

VMS Usage: **process_id**
 type: **longword (unsigned)**
 access: **modify**
 mechanism: **by reference**

Process identification (PID) of the process about which \$GETJPI is to return information. The *pidadr* argument is the address of a longword containing the PID.

System Service Descriptions

\$GETJPI

If **pidadr** is specified as -1, \$GETJPI assumes a wildcard operation and returns the requested information for each process on the system that it has the privilege to access, one process per call. To perform a wildcard operation, the user must call \$GETJPI in a loop, testing for the condition value SS\$_NOMOREPROC after each call and exiting the loop when SS\$_NOMOREPROC is returned.

For more information see Table SYS-4, which appears in the "Description" section.

prcnam

VMS Usage: **process_name**
type: **character-coded text string**
access: **read only**
mechanism: **by descriptor-fixed length string descriptor**

Name of the process about which \$GETJPI is to return information. The **prcnam** argument is the address of a character string descriptor pointing to this name string.

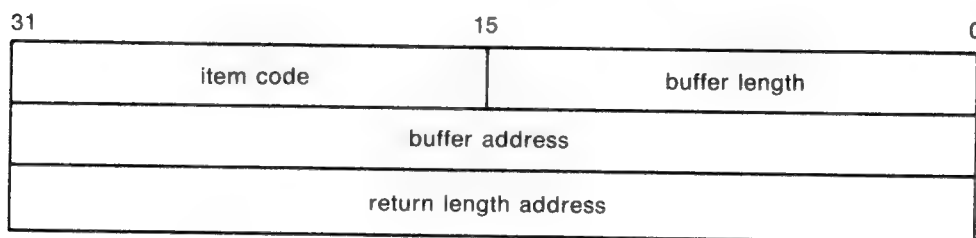
The process name string must contain from 1 to 15 characters and must correspond exactly to the process name; no trailing blanks or abbreviations are permitted.

The **prcnam** argument may be used only if the process identified by **prcnam** has the same UIC group number as the calling process. If the process has a different group number, \$GETJPI returns no information. To obtain information about processes in other groups, the **pidadr** argument must be used.

itmlst

VMS Usage: **item_list_3**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Item list specifying which information about the process(es) is to be returned. The **itmlst** argument is the address of a list of item descriptors, each of which describes an item of information. The list of item descriptors is terminated by a longword of 0. The following diagram depicts the format of a single item descriptor.



ZK-1705-84

\$GETJPI Item Descriptor Fields

buffer length

A word containing a user-supplied integer specifying the length (in bytes) of the buffer in which \$GETJPI is to write the information. The length of the buffer needed depends upon the item code specified in the **item code** field

System Service Descriptions

\$GETJPI

of the item descriptor. If the value of **buffer length** is too small, \$GETJPI truncates the data.

item code

A word containing a user-supplied symbolic code specifying the item of information that \$GETJPI is to return. These codes are defined by the \$JPIDEF macro. Each item code is described below.

buffer address

A longword containing the user-supplied address of the buffer in which \$GETJPI is to write the information.

return length address

A longword containing the user-supplied address of a word in which \$GETJPI writes the length in bytes of the information it actually returned.

\$GETJPI Item Codes

JPI\$_ACCOUNT

When JPI\$_ACCOUNT is specified, \$GETJPI returns the process's account name, which is an 8-byte string, filled with trailing blanks if necessary.

JPI\$_APTCNT

When JPI\$_APTCNT is specified, \$GETJPI returns the process's active page table count, which is a longword decimal value.

JPI\$_ASTACT

When JPI\$_ASTACT is specified, \$GETJPI returns the names of the access modes having active ASTs. This information is returned in a longword bit vector. When bit 0 is set, an active kernel mode AST exists; bit 1, an executive mode AST; bit 2, a supervisor mode AST; and bit 3, a user mode AST.

JPI\$_ASTCNT

When JPI\$_ASTCNT is specified, \$GETJPI returns a count of the remaining AST quota, which is a longword decimal value.

JPI\$_ASTEN

When JPI\$_ASTEN is specified, \$GETJPI returns the names of the access modes having ASTs enabled. This information is returned in a longword bit vector. When bit 0 is set, kernel mode has ASTs enabled; bit 1, executive mode; bit 2, supervisor mode; and bit 3, user mode.

JPI\$_ASTLM

When JPI\$_ASTLM is specified, \$GETJPI returns the process's AST limit quota, which is a longword decimal value.

JPI\$_AUTHPRI

When JPI\$_AUTHPRI is specified, \$GETJPI returns the process's authorized base priority, which is a longword decimal value. The authorized base priority is the highest priority a process without ALTPRI privilege can attain by means of the \$SETPRI service.

JPI\$_AUTHPRIV

When JPI\$_AUTHPRIV is specified, \$GETJPI returns the privileges that the process is authorized to enable. These privileges are returned in a quadword privilege mask and are defined by the \$PRVDEF macro.

System Service Descriptions

\$GETJPI

JPI\$_BIOCNT

When JPI\$_BIOCNT is specified, \$GETJPI returns a count of the remaining buffered I/O quota, which is a longword decimal value.

JPI\$_BIOLM

When JPI\$_BIOLM is specified, \$GETJPI returns the process's buffered I/O limit quota, which is a longword decimal value.

JPI\$_BUFIO

When JPI\$_BUFIO is specified, \$GETJPI returns a count of the process's buffered I/O operations, which is a longword decimal value.

JPI\$_BYTCNT

When JPI\$_BYTCNT is specified, \$GETJPI returns the process's remaining buffered I/O byte count quota, which is a longword decimal value.

JPI\$_BYTLM

When JPI\$_BYTLM is specified, \$GETJPI returns the process's buffered I/O byte count limit quota, which is a longword decimal value.

JPI\$_CHAIN

When JPI\$_CHAIN is specified, \$GETJPI processes another item list immediately after processing the current one. The **buffer address** field in the item descriptor specifies the address of the next item list to be processed. The JPI\$_CHAIN item code must be the last item code specified in the item list.

JPI\$_CLNAME

When JPI\$_CLNAME is specified, \$GETJPI returns the name of the command language interpreter that the process is currently using. Because the CLI name can include up to 39 characters, the **buffer length** field in the item descriptor should specify 39 bytes.

JPI\$_CPULIM

When JPI\$_CPULIM is specified, \$GETJPI returns the process's CPU time limit, which is a longword decimal value.

JPI\$_CPUTIM

When JPI\$_CPUTIM is specified, \$GETJPI returns the process's accumulated CPU time in 10-millisecond ticks, which is a longword decimal value.

JPI\$_CREPRC_FLAGS

When JPI\$_CREPRC_FLAGS is specified, \$GETJPI returns the flags specified by the **flags** argument in the \$CREPRC call that created the process. The flags are returned as a longword bit vector.

JPI\$_CURPRIV

When JPI\$_CURPRIV is specified, \$GETJPI returns the process's current privileges. These privileges are returned in a quadword privilege mask and are defined by the \$PRVDEF macro.

JPI\$_DFPFC

When JPI\$_DFPFC is specified, \$GETJPI returns the process's default page fault cluster size, which is a longword decimal value.

System Service Descriptions

\$GETJPI

JPI\$_DFWSCNT

When JPI\$_DFWSCNT is specified, \$GETJPI returns the process's default working set size, which is a longword decimal value.

JPI\$_DIOCNT

When JPI\$_DIOCNT is specified, \$GETJPI returns the process's remaining direct I/O quota, which is a longword decimal value.

JPI\$_DIOLM

When JPI\$_DIOLM is specified, \$GETJPI returns the process's direct I/O quota limit, which is a longword decimal value.

JPI\$_DIRIO

When JPI\$_DIRIO is specified, \$GETJPI returns a count of the process's direct I/O operations, which is a longword decimal value.

JPI\$_EFCS

When JPI\$_EFCS is specified, \$GETJPI returns the state of the process's local event flags 0 through 31 as a longword bit vector.

JPI\$_EFCU

When JPI\$_EFCU is specified, \$GETJPI returns the state of the process's local event flags 32 through 63 as a longword bit vector.

JPI\$_EFWM

When JPI\$_EFWM is specified, \$GETJPI returns the process's event flag wait mask, which is a longword bit vector.

JPI\$_ENQCNT

When JPI\$_ENQCNT is specified, \$GETJPI returns the process's remaining lock request quota, which is a longword decimal value.

JPI\$_ENQLM

When JPI\$_ENQLM is specified, \$GETJPI returns the process's lock request quota, which is a longword decimal value.

JPI\$_EXCVEC

When JPI\$_EXCVEC is specified, \$GETJPI returns the address of a list of exception vectors for the process. Each exception vector in the list is a longword. There are eight vectors in the list: these are, in order, a primary and a secondary vector for kernel mode access, for executive mode access, for supervisor mode access, and for user mode access.

\$GETJPI cannot return this information for any process other than the calling process; if this item code is specified and the process is not the calling process, \$GETJPI returns a zero in the buffer.

JPI\$_FILCNT

When JPI\$_FILCNT is specified, \$GETJPI returns the process's remaining open file quota, which is a longword decimal value.

JPI\$_FILLM

When JPI\$_FILLM is specified, \$GETJPI returns the process's open file limit quota, which is a longword value.

System Service Descriptions

\$GETJPI

JPI\$_FINALEXC

When JPI\$_FINALEXC is specified, \$GETJPI returns the address of a list of final exception vectors for the process. Each exception vector in the list is a longword. There are four vectors in the list, one for each access mode, in the order kernel, executive, supervisor, and user.

\$GETJPI cannot return this information for any process other than the calling process; if this item code is specified and the process is not the calling process, \$GETJPI returns a zero in the buffer.

JPI\$_FREPOVA

When JPI\$_FREPOVA is specified, \$GETJPI returns the address of the first free page at the end of the process's program region (P0 space).

JPI\$_FREP1VA

When JPI\$_FREP1VA is specified, \$GETJPI returns the address of the first free page at the end of the process's control region (P1 space).

JPI\$_FREPTCNT

When JPI\$_FREPTCNT is specified, \$GETJPI returns the number of pages that the process has available for virtual memory expansion. This value is a longword decimal value.

JPI\$_GPGCNT

When JPI\$_GPGCNT is specified, \$GETJPI returns the process's global page count in the working set, which is a longword decimal value.

JPI\$_GRP

When JPI\$_GRP is specified, \$GETJPI returns the group number of the process's UIC. This is a longword decimal value.

JPI\$_IMAGECOUNT

When JPI\$_IMAGECOUNT is specified, \$GETJPI returns, as a longword decimal value, the number of images that have been run down for the process.

JPI\$_IMAGNAME

When JPI\$_IMAGNAME is specified, \$GETJPI returns, as a character string, the name of the current image file. Since the name of the image file can include up to 39 characters, the buffer length field in the item descriptor should specify 39(bytes).

JPI\$_IMAGPRIV

When JPI\$_IMAGPRIV is specified, \$GETJPI returns a quadword mask of the privileges with which the current image was installed. If the current image was not installed, \$GETJPI returns a 0 in the buffer.

JPI\$_JOBPRCNT

When JPI\$_JOBPRCNT is specified, \$GETJPI returns the total number of subprocesses owned by the job, which is a longword decimal value.

JPI\$_JOBTYP

When JPI\$_JOBTYP is specified, \$GETJPI returns the execution mode of the process at the root of the job tree, which is a longword decimal value. The symbolic name and value for each execution mode are listed in the following table; the symbolic names are defined by the \$JPIDEF macro.

System Service Descriptions

\$GETJPI

Mode Name	Value
JPI\$C_DETACHED	0
JPI\$C_NETWORK	1
JPI\$C_BATCH	2
JPI\$C_LOCAL	3
JPI\$C_DIALUP	4
JPI\$C_REMOTE	5

JPI\$_LOGINTIM

When JPI\$_LOGINTIM is specified, \$GETJPI returns the time at which the process was created, which is a standard 64-bit absolute time.

JPI\$_MASTER_PID

When JPI\$_MASTER_PID is specified, \$GETJPI returns the process identification (PID) of the process's parent process in the job. The PID is a longword hexadecimal value.

JPI\$_MAXDETACH

When JPI\$_MAXDETACH is specified, \$GETJPI returns the maximum number of detached processes allowed for the user name of the user who owns the process specified in the call to \$GETJPI. This limit is set in the UAF record of the user. The number is returned as a word decimal value. A value of 0 means that there is no limit on the number of detached processes for that user name.

JPI\$_MAXJOBS

When JPI\$_MAXJOBS is specified, \$GETJPI returns the maximum number of active processes allowed for the user name of the user who owns the process specified in the call to \$GETJPI. This limit is set in the UAF record of the user. The number is returned as a word decimal value. A value of 0 means that there is no limit on the number of active processes for that user name.

JPI\$_MEM

When JPI\$_MEM is specified, \$GETJPI returns the member number of the process's UIC, which is a longword decimal value.

JPI\$_MODE

When JPI\$_MODE is specified, \$GETJPI returns the mode of the process, which is a longword decimal value. The symbolic name and value for each mode are listed in the following table; the symbolic names are defined by the \$JPIDEF macro.

Mode Name	Value
JPI\$K_OTHER	0
JPI\$K_NETWORK	1
JPI\$K_BATCH	2
JPI\$K_INTERACTIVE	3

JPI\$_MSGMASK

When JPI\$_MSGMASK is specified, \$GETJPI returns the process's default message mask, which is a longword bit mask.

System Service Descriptions

\$GETJPI

JPI\$_OWNER

When JPI\$_OWNER is specified, \$GETJPI returns the process identification (PID) of the process that created the specified process. The PID is a longword hexadecimal value.

JPI\$_PAGEFLTS

When JPI\$_PAGEFLTS is specified, \$GETJPI returns the total number of page faults incurred by the process. This is a longword decimal value.

JPI\$_PAGFILCNT

When JPI\$_PAGFILCNT is specified, \$GETJPI returns the process's remaining paging file quota, which is a longword decimal value.

JPI\$_PAGFILLOC

When JPI\$_PAGFILLOC is specified, \$GETJPI returns the address of the process's paging file; the fourth byte of this address is the process's page file index.

JPI\$_PGFLQUOTA

When JPI\$_PGFLQUOTA is specified, \$GETJPI returns the process's paging file quota, which is a longword decimal value.

JPI\$_PHDFLAGS

When JPI\$_PHDFLAGS is specified, \$GETJPI returns the process header flags as a longword bit vector.

JPI\$_PID

When JPI\$_PID is specified, \$GETJPI returns the process identification (PID) of the process. The PID is a longword hexadecimal value.

JPI\$_PPGCNT

When JPI\$_PPGCNT is specified, \$GETJPI returns the number of pages the process has in the working set. This is a longword decimal value.

JPI\$_PRCNT

When JPI\$_PRCNT is specified, \$GETJPI returns, as a longword decimal value, the number of subprocesses created by the process. The number returned by JPI\$_PRCNT does not include any subprocesses created by subprocesses of the process named in the **procnam** argument.

JPI\$_PRCLM

When JPI\$_PRCLM is specified, \$GETJPI returns the process's subprocess quota, which is a longword decimal value.

JPI\$_PRCNAM

When JPI\$_PRCNAM is specified, \$GETJPI returns, as a character string, the name of the process. Since the process name can include up to 15 characters, the buffer length field of the item descriptor should specify 15 (bytes).

JPI\$_PRI

When JPI\$_PRI is specified, \$GETJPI returns the process's current priority, which is a longword decimal value.

JPI\$_PRIB

When JPI\$_PRIB is specified, \$GETJPI returns the process's base priority, which is a longword decimal value.

JPI\$_PROC_INDEX

When JPI\$_PROC_INDEX is specified \$GETJPI returns, as a longword decimal value, the process index number of the process. The process index number is a number between 1 and the sysgen parameter MAXPROCESSCNT, which identifies the process. Although process index numbers are reassigned to different processes over time, at any one instant, each process in the system has a unique process index number.

The user may want to use the process index number as an index into system global sections. Because the process index number is unique for each process, the use of it as an index into system global sections guarantees no collisions with other system processes accessing those sections.

The process index is intended to serve users who formerly used the low order word of the PID as an index number.

JPI\$_PROCPRIV

When JPI\$_PROCPRIV is specified, \$GETJPI returns the process's default privileges in a quadword bit mask.

JPI\$_SHRFILLM

When JPI\$_SHRFILLM is specified, \$GETJPI returns the maximum number of open shared files allowed for the job to which the process specified in the call to \$GETJPI belongs. This limit is set in the UAF record of the user who owns the process. The number is returned as a word decimal value. A value of 0 means that there is no limit on the number of open shared files for that job.

JPI\$_SITESPEC

When JPI\$_SITESPEC is specified, \$GETJPI returns the per-process, site-specific longword, which is a longword decimal value.

JPI\$_STATE

When JPI\$_STATE is specified, \$GETJPI returns the process's state, which is a longword decimal value. Each state has a symbolic representation. If the process is currently executing, its state is always SCH\$K_CUR. The \$STATEDEF macro defines the following symbols, which identify the various possible states:

State	Description
SCH\$_CEF	Common event flag wait
SCH\$_COM	Computable
SCH\$_COMO	Computable, out of balance set
SCH\$_CUR	Current process
SCH\$_COLPG	Collided page wait
SCH\$_FPG	Free page wait
SCH\$_HIB	Hibernate wait
SCH\$_HIBO	Hibernate wait, out of balance set
SCH\$_LEF	Local event flag wait
SCH\$_LEFO	Local event flag wait, out of balance set
SCH\$_MWAIT	Mutex and miscellaneous resource wait
SCH\$_PFW	Page fault wait

System Service Descriptions

\$GETJPI

State	Description
SCH\$_SUSP	Suspended
SCH\$_SUSPO	Suspended, out of balance set

JPI\$_STS

When JPI\$_STS is specified, \$GETJPI returns the process's status flags, which are contained in a longword bit vector. Listed below are the symbols for these flags, which are defined by the \$PCBDEF macro.

Flag	Description
PCB\$_ASTPEN	AST pending
PCB\$_BATCH	Process is a batch job
PCB\$_DELPEN	Delete pending
PCB\$_DISAWS	Disable automatic working set adjustment
PCB\$_FORCPEN	Force exit pending
PCB\$_HIBER	Hibernate after initial image activate
PCB\$_INQUAN	Initial quantum in progress
PCB\$_INTER	Process is an interactive job
PCB\$_LOGIN	Login without reading authorization file
PCB\$_NETWRK	Process is a network connect object
PCB\$_NOACNT	No accounting for process
PCB\$_NODELET	No delete
PCB\$_PHDRES	Process header resident
PCB\$_PSWAPM	Process swap mode (1=noswap)
PCB\$_PWRAST	Power fail AST
PCB\$_RECOVER	Process can recover locks
PCB\$_RES	Resident, in balance set
PCB\$_RESPEN	Resume pending, skip suspend
PCB\$_SSFEXC	System service exception enable (kernel)
PCB\$_SSFEXCE	System service exception enable (exec)
PCB\$_SSFEXCS	System service exception enable (super)
PCB\$_SSFEXCU	System service exception enable (user)
PCB\$_SSRWAIT	System service resource wait disable
PCB\$_SUSPEN	Suspend pending
PCB\$_SWPVBN	Write for swap VBN in progress
PCB\$_WAKEPEN	Wake pending, skip hibernate
PCB\$_WALL	Wait for all events in mask

JPI\$_SWPFILLOC

When JPI\$_SWPFILLOC is specified, \$GETJPI returns the location of the process's swapping file, which is a longword hexadecimal value. The fourth byte of this value identifies a specific swapping file; the lower three bytes contain the VBN within the swapping file.

JPI\$_TABlename

When JPI\$_TABlename is specified, \$GETJPI returns the file specification of the process's current command language interpreter (CLI) table. Since the file specification can include up to 255 characters, the buffer length field in the item descriptor should specify 255 (bytes).

JPI\$_TERmINAL

When JPI\$_TERmINAL is specified, \$GETJPI returns, for interactive users, the process's login terminal name as a character string. Since the terminal name can include up to 7 characters, the buffer length field in the item descriptor should specify 7 (bytes).

JPI\$_TMBU

When JPI\$_TMBU is specified, \$GETJPI returns the termination mailbox unit number, which is a longword decimal value.

JPI\$_TQCNT

When JPI\$_TQCNT is specified, \$GETJPI returns the process's remaining timer queue entry quota, which is a longword decimal value.

JPI\$_TQLM

When JPI\$_TQLM is specified, \$GETJPI returns the process's limit on timer queue entries, which is a longword decimal value.

JPI\$_UAF_FLAGS

When JPI\$_UAF_FLAGS is specified, \$GETJPI returns the UAF flags from the UAF record of the user who owns the process. The flags are returned as a longword bit vector.

JPI\$_UIC

When JPI\$_UIC is specified, \$GETJPI returns the process's UIC in the standard longword format.

JPI\$_USERNAME

When JPI\$_USERNAME is specified, \$GETJPI returns the process's user name as a 12-byte string. \$GETJPI fills out the 12 bytes with trailing blanks if the name is less than 12 bytes.

JPI\$_VIRTPEAK

When JPI\$_VIRTPEAK is specified, \$GETJPI returns the process's peak virtual address size as a longword decimal value.

JPI\$_VOLUMES

When JPI\$_VOLUMES is specified, \$GETJPI returns the number of volumes that the process currently has mounted, which is a longword decimal value.

JPI\$_WSAUTH

When JPI\$_WSAUTH is specified, \$GETJPI returns the process's maximum authorized working set size as a longword decimal value.

JPI\$_WSAUTHEXT

When JPI\$_WSAUTHEXT is specified, \$GETJPI returns the process's maximum authorized working set extent as a longword decimal value.

JPI\$_WSEXTENT

When JPI\$_WSEXTENT is specified, \$GETJPI returns the process's current working set extent as a longword decimal value.

System Service Descriptions

\$GETJPI

JPI\$_WSPEAK

When JPI\$_WSPEAK is specified, \$GETJPI returns the process's peak working set size as a longword decimal value.

JPI\$_WSQUOTA

When JPI\$_WSQUOTA is specified, \$GETJPI returns the process's working set size quota as a longword decimal value.

JPI\$_WSSIZE

When JPI\$_WSSIZE is specified, \$GETJPI returns the process's current working set size as a longword decimal value.

iosb

VMS Usage: **io_status_block**
type: **quadword (unsigned)**
access: **write only**
mechanism: **by reference**

I/O status block which is to receive the final completion status. The **iosb** is the address of the quadword I/O status block.

When the **iosb** argument is specified, \$GETJPI sets the quadword to zero upon request initiation. Upon request completion, a condition value is returned to the first longword; the second longword is reserved to DIGITAL.

Though this argument is optional, DIGITAL strongly recommends that you specify it for the following reasons.

- If you are using an event flag to signal the completion of the service, you can test the I/O status block for a condition value to be sure that the event flag was not set by an event other than service completion.
- If you are using the \$SYNCH service to synchronize completion of the service, the I/O status block is a required argument for \$SYNCH.
- The condition value returned in R0 and the condition value returned in the I/O status block provide information about different aspects of the call to the \$GETJPI service. The condition value returned in R0 gives you information about the success or failure of the service call itself; the condition value returned in the I/O status block gives you information about the success or failure of the service operation. Therefore, to accurately assess the success or failure of the call to \$GETJPI, you must check the condition values returned in both R0 and the I/O status block.

Refer to Section 2.5 for more information about system service completion.

astadr

VMS Usage: **ast_procedure**
type: **procedure entry mask**
access: **call without stack unwinding**
mechanism: **by reference**

AST service routine to be executed when \$GETJPI completes. The **astadr** is the address of the entry mask of this routine.

If **astadr** is specified, the AST routine will execute at the same access mode as the caller of the \$GETJPI service.

System Service Descriptions

\$GETJPI

astprm

VMS Usage: **user_arg**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

AST parameter to be passed to the AST service routine specified by the **astadr** argument. The **astprm** argument is the longword parameter.

DESCRIPTION

The calling process must have GROUP privilege to obtain information about other processes with the same group UIC number as the calling process.

The calling process must have WORLD privilege to obtain information about other processes on the system that are not in the same group as the calling process.

Getting information about another process is an asynchronous operation because the information may be contained in the other process's virtual address space and the process might have a lower priority or might be currently swapped out of the balance set. To allow your program to overlap other functions with the time needed to schedule the other process for execution or swap it into the balance set, \$GETJPI returns immediately after it has queued its information-gathering request to the other process.

It is better to specify a process to \$GETJPI by using the **pidadr** argument instead of the **prcnam** argument. This is true for the following two reasons:

- The **pidadr** argument may be used to identify any process in the system, whereas the **prcnam** argument can be used only to identify processes that have the same UIC group number as the caller of \$GETJPI.
- \$GETJPI executes faster when **pidadr** rather than **prcnam** is used. When **prcnam** is specified, \$GETJPI must search a table of process names and UICs for an entry that contains the specified process name and the UIC group number of the calling process; this search is unnecessary when **pidadr** is used.

With respect to privileges, the calling process must have GROUP privilege to obtain information about any process (except itself) in the same group. The calling process must have WORLD privilege (and must use **pidadr** rather than **prcnam**) to obtain information about a process with a different UIC group number.

Table SYS-4 shows how \$GETJPI operates given various values for the **prcnam** and **pidadr** arguments.

Table SYS-4 Process Identification in \$GETJPI

Process Name Specified?	Process ID Specified?	Process ID Contains	Action by \$GETJPI
No	No	—	The process identification of the calling process is used, and the process identification is not returned.
No	Yes	0	The process identification of the calling process is used and returned.
No	Yes	Process ID	The process identification is used and returned.

\$GETJPI

Process Name Specified?	Process ID Specified?	Process ID Contains	Action by \$GETJPI
Yes	No	—	The process name is used, and the process identification is not returned.
Yes	Yes	0	The process name is used, and the process identification is returned.
Yes	Yes	Process ID	The process identification is used and returned, and the process name is ignored.

CONDITION VALUES RETURNED		
SS\$_NORMAL		Successful completion.
SS\$_BADPARAM		The item list contains an invalid identifier.
SS\$_ACCVIO		The item list cannot be read by the caller, or the buffer length or buffer cannot be written by the caller.
SS\$_IVLOGNAM		The process name string has a length of 0 or has more than 15 characters.
SS\$_NOMOREPROC		Warning. In a wildcard operation, \$GETJPI found no more processes.
SS\$_NONEXPR		Warning. The specified process does not exist, or an invalid process identification was specified.
SS\$_NOPRIV		The process does not have the privilege to obtain information about the specified process.
SS\$_SUSPENDED		The specified process is suspended or in a miscellaneous wait state, and the requested information cannot be obtained.

CONDITION VALUES RETURNED IN THE I/O STATUS BLOCK	Same as those returned in R0.
--	-------------------------------

EXAMPLE

```

; Define $JPIDEF
; Define $GETJPI item codes
;
PID: .LONG -1 ; "Wild card" PID
ITEMS: .WORD 12 ; Size of username buffer
; Username item code
.WORD JPI$_USERNAME
.ADDRESS -
UNAME ; Address of username buffer
.ADDRESS -
UNAMES ; Address to return username size
.LONG 0 ; End of list
UNAME: .BLKB 12 ; Username buffer
UNAMES: .BLKL 1 ; Username size buffer
IOSEB: .BLKQ 1 ; Completion status

```

System Service Descriptions

\$GETJPI

```
;
START:  .WORD  0

;
LOOP:   $GETJPI_S -
        EFN=#1, -
        PIDADR=PID, -
        ITMLST=ITEMS, -
        IOSB=IOSB
        BLBS  RO, WAIT      ; If success, continue
        CMPW  RO, $SS$_NOPRIV ; No privilege to get info on process?
        BEQL  LOOP          ; If no priv, try next process
        CMPW  RO, $SS$_SUSPENDED ; Process suspended?
        BEQL  LOOP          ; If yes, try next process
        CMPW  RO, $SS$_NOMOREPROC ; No more processes?
        BEQL  DONE          ; If yes, all done
        BSBW  ERROR         ; Else, error

;
WAIT:   $WAITFR_S -
        EFN=#1      ; Wait for information
        MOVZWL IOSB, RO ; Get completion status
        BSBW  ERROR  ; Check for errors
        BSBW  DISPLAY_NAME ; Display the name
        BRB   LOOP
```

The above example shows a segment of a program used to obtain the user name of every process for which the caller has the privilege to obtain information.

System Service Descriptions

\$GETJPIW

\$GETJPIW—Get Job/Process Information and Wait

The Get Job/Process Information and Wait service returns information about one or more processes on the system.

The \$GETJPIW service completes synchronously; that is, it returns to the caller with the requested information.

For asynchronous completion, use the Get Job/Process Information (\$GETJPI) service; \$GETJPI returns to the caller after queuing the information request, without waiting for the information to be returned.

In all other respects, \$GETJPIW is identical to \$GETJPI. Refer to the documentation of \$GETJPI for all other information about the \$GETJPIW service.

For additional information about system service completion, refer to the Synchronize (\$SYNCH) service and to Section 2.5.

FORMAT

SYSS\$GETJPIW *[efn],[pidadr],[prcnam],itmlst[,iosb]
[,astadr],[astprm]*

\$GETLKI—Get Lock Information

The Get Lock Information service returns information about the lock data base on a VAX/VMS system.

The \$GETLKI service completes asynchronously; that is, it returns to the caller after queuing the information request, without waiting for the information to be returned.

For synchronous completion, use the Get Lock Information and Wait (\$GETLKIW) service; \$GETLKIW returns to the caller with the requested information. In all other respects, \$GETLKIW is identical to \$GETLKI.

For additional information about system service completion, refer to the Synchronize (\$SYNCH) service and to Section 2.5.

The \$GETLKI, \$GETLKIW, \$ENQ (Enqueue Lock Request), \$ENQW (Enqueue Lock Request and Wait), and \$DEQ (Dequeue Lock Request) services together provide the user interface to the VAX/VMS lock management facility. Refer to the descriptions of these other services and to Section 12 in Part I for additional information about lock management.

FORMAT **SYSS\$GETLKI** *[efn]* ,*[lkidadr]* ,*itmlst* ,*[iosb]* ,*[astadr]* ,*[astprm]* ,*[nullarg]*

RETURNS

VMS Usage: **cond_value**
 type: **longword (unsigned)**
 access: **write only**
 mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS *efn*

VMS Usage: **ef_number**
 type: **longword (unsigned)**
 access: **read only**
 mechanism: **by value**

Number of the event flag to be set when \$GETLKI completes. The *efn* argument is a longword containing this number. If *efn* is not specified, \$GETLKI sets event flag 0.

lkidadr

VMS Usage: **lock_id**
 type: **longword (unsigned)**
 access: **modify**

System Service Descriptions

\$GETLKI

mechanism: **by reference**

Lock identification (lock id) for the lock about which information is to be returned. The lock id is the second longword in the lock status block, which was created when the lock was granted. The *lkidadr* argument is the address of this longword.

If the value specified by *lkidadr* is 0 or -1, \$GETLKI assumes a wildcard operation and returns information about each lock to which the calling process has access, one lock per call.

The \$GETLKI service must have read/write access to the lock id.

itmlst

VMS Usage: *item_list_3*

type: **longword (unsigned)**

access: **read only**

mechanism: **by reference**

Item list specifying the lock information that \$GETLKI is to return. The *itmlst* argument is the address of a list of item descriptors, each of which describes an item of information. The list of item descriptors is terminated by a longword of 0.

31	15	0
item code		buffer length
buffer address		
return length address		

ZK-1705-84

\$GETLKI Item Descriptor Fields

buffer length

A word containing a user-supplied integer specifying the length (in bytes) of the buffer in which \$GETLKI is to write the information. The length of the buffer needed depends upon the item code specified in the *item code* field of the item descriptor. If the value of *buffer length* is too small, \$GETLKI truncates the data and returns the success condition value *SS\$_NORMAL*.

item code

A word containing a user-supplied symbolic code specifying the item of information that \$GETLKI is to return. These codes are defined by the \$LKIDEF macro. Each item code is described in the "\$GETLKI Item Codes" section.

buffer address

A longword containing the user-supplied address of the buffer in which \$GETLKI is to write the information.

return length address

A longword containing the user-supplied address of a longword in which \$GETLKI writes return length information. This longword contains three bit fields:

System Service Descriptions

\$GETLKI

Bits	Description
0 to 15	In this field \$GETLKI writes the length in bytes of the data actually written to the buffer specified by the buffer address field in the item descriptor.
16 to 30	\$GETLKI uses this field only when the item code field of the item descriptor specifies LKIS_BLOCKEDBY, LKIS_BLOCKING, or LKIS_LOCKS, each of which requests information about a list of locks. \$GETLKI writes in this field the length in bytes of the information returned for a single lock in the list. The idea here is that the user can divide this length into the total length returned for all locks (bits 0 to 15) to determine the number of locks located by that item code request.
31	\$GETLKI sets this bit if the user-supplied buffer length argument specifies too small a buffer to contain the information returned. Note that in such a case \$GETLKI will return the SS\$ _NORMAL condition value in R0. Therefore, to locate any faulty item descriptor, it is necessary to check the state of bit 31 in the longword specified by the return length field of each item descriptor.

\$GETLKI Item Codes

LKIS_BLOCKEDBY

When LKIS_BLOCKEDBY is specified, \$GETLKI returns information about all locks that are currently blocked by the lock specified by **lkidadr**. \$GETLKI returns eight items of information about each blocked lock.

The eight items in the buffer may be referred to by the following symbolic names, which are defined by the \$LKIDEF macro.

Symbolic Name	Description
LKIS\$ _LOCKID	Lock id of the blocked lock on the system maintaining the resource (4 bytes)
LKIS\$ _PID	Process id (PID) of the process that took out the blocked lock (4 bytes)
LKIS\$ _SYSID	Cluster system identifier (CSID) of the VAX node that is maintaining the resource that is locked by the blocked lock (4 bytes)
LKIS\$B _RQMODE	Lock mode requested for the blocked lock; this lock mode was specified by the lkmode argument in the call to \$ENQ (1 byte)
LKIS\$B _GRMODE	Lock mode granted to the blocked lock; this lock mode is written to the lock value block (1 byte)
LKIS\$B _QUEUE	Name of the queue on which the blocked lock currently resides (1 byte)
LKIS\$ _REMLKID	Lock id of the lock on the system where the lock was requested (4 bytes)
LKIS\$ _REMSYSID	System id of the system where the lock was requested (4 bytes)

The values that \$GETLKI may write into LKIS\$B _RQMODE, LKIS\$B _GRMODE, and LKIS\$B _QUEUE items have symbolic names; these symbolic names specify the six lock modes and the three types of queue in which a lock may reside. The Description section list describes these names.

System Service Descriptions

\$GETLKI

Thus, the buffer specified by the **buffer address** field in the item descriptor will contain the above eight items of information, repeated in sequence, for each blocked lock.

The length of the information returned for each blocked lock is returned in bits 16 to 30 of the longword specified by the **return length address** field in the item descriptor, while the total length of information returned for all blocked locks is returned in bits 0 to 15. Therefore, to determine the number of blocked locks, divide the value of bits 16 to 30 into the value of bits 0 to 15.

LKI\$_BLOCKING

When LKI\$_BLOCKING is specified, \$GETLKI returns information about all locks that are currently blocking the lock specified by **lkidadr**. \$GETLKI returns eight items of information about each blocking lock.

The eight items in the buffer may be referred to by the following symbolic names, which are defined by the \$LKIDEF macro.

Symbolic Name	Description
LKI\$_LOCKID	Lock id of the blocked lock on the system maintaining the resource (4 bytes)
LKI\$_PID	Process id (PID) of the process that took out the blocking lock (4 bytes)
LKI\$_SYSID	Cluster system identifier (CSID) of the VAX node that is maintaining the resource that is locked by the blocking lock (4 bytes)
LKI\$_RQMODE	Lock mode requested for the blocking lock; this lock mode was specified by the lkmode argument in the call to \$ENQ (1 byte)
LKI\$_GRMODE	Lock mode granted to the blocking lock; this lock mode is written to the lock value block (1 byte)
LKI\$_QUEUE	Name of the queue on which the blocking lock currently resides (1 byte)
LKI\$_REMLKID	Lock id of the lock on the system where the lock was requested (4 bytes)
LKI\$_REMSYSID	System id of the system where the lock was requested (4 bytes)

The values that \$GETLKI may write into LKI\$_RQMODE, LKI\$_GRMODE, and LKI\$_QUEUE items have symbolic names; these symbolic names specify the six lock modes and the three types of queue in which a lock may reside. The Description section lists these names.

Thus, the buffer specified by the **buffer address** field in the item descriptor will contain the above eight items of information, repeated in sequence, for each blocking lock.

The length of the information returned for each blocking lock is returned in bits 16 to 30 of the longword specified by the **return length address** field in the item descriptor, while the total length of information returned for all blocking locks is returned in bits 0 to 15. Therefore, to determine the number of blocking locks, divide the value of bits 16 to 30 into the value of bits 0 to 15.

System Service Descriptions

\$GETLKI

LKIS_LCKCOUNT

When LKIS_LCKCOUNT is specified, \$GETLKI returns the total number of locks that have been granted on the resource associated with the current lock. This information is useful when one wishes to modify or delete the resource.

The **buffer length** field in the item descriptor should specify 4 (bytes).

LKIS_LCKREFCNT

When LKIS_LCKREFCNT is specified, \$GETLKI returns the number of locks that have this lock as a parent lock. When these locks were created, the **parid** argument in the call to \$ENQ or \$ENQW specified the lock id of this lock.

The **buffer length** field in the item descriptor should specify 4 (bytes).

LKIS_LOCKID

When LKIS_LOCKID is specified, \$GETLKI returns the lock id of the current lock. The current lock is the lock specified by the **lkidadr** argument unless **lkidadr** is specified as -1 or 0, which indicates a wildcard operation. Thus, this item code is usually only specified in wildcard operations where it is useful to know the lock ids of the locks that \$GETLKI has discovered in the wildcard operation.

The lock id is a longword value, so the **buffer length** field in the item descriptor should specify 4 (bytes).

LKIS_LOCKS

When LKIS_LOCKS is specified, \$GETLKI returns information about all locks on the resource associated with the lock specified by **lkidadr**. These locks are the sum of blocking locks and blocked locks.

The eight items in the buffer may be referred to by the following symbolic names, which are defined by the \$LKIDEF macro.

Symbolic Name	Description
LKISL_LOCKID	Lock id of the blocked lock on the system maintaining the resource (4 bytes)
LKISL_PID	Process id (PID) of the process that took out the lock (4 bytes)
LKISL_SYSID	Cluster system identifier (CSID) of the VAX node that is maintaining the resource that is locked by the lock (4 bytes)
LKISB_RQMODE	Lock mode requested for the lock; this lock mode was specified by the lkmode argument in the call to \$ENQ (1 byte)
LKISB_GRMODE	Lock mode granted to the lock; this lock mode is written to the lock value block (1 byte)
LKISB_QUEUE	Name of the queue on which the lock currently resides (1 byte)
LKISL_REMLKID	Lock id of the lock on the system where the lock was requested (4 bytes)
LKISL_REMSYSID	System id of the system where the lock was requested (4 bytes)

The values that \$GETLKI may write into LKISB_RQMODE, LKISB_GRMODE, and LKISB_QUEUE items have symbolic names; these symbolic

System Service Descriptions

\$GETLKI

names specify the six lock modes and the three types of queue in which a lock may reside. The Description section lists these names.

Thus, the buffer specified by the **buffer address** field in the item descriptor will contain the above eight items of information, repeated in sequence, for each lock.

The length of the information returned for each lock is returned in bits 16 to 30 of the longword specified by the **return length address** field in the item descriptor, while the total length of information returned for all locks is returned in bits 0 to 15. Therefore, to determine the number of locks, divide the value of bits 16 to 30 into the value of bits 0 to 15.

LKIS_NAMESPACE

When LKIS_NAMESPACE is specified, \$GETLKI returns information about the resource name space. This information is contained in a longword consisting of four bit fields; therefore, the **buffer length** field in the item descriptor should specify 4 (bytes).

Each of the four bit fields may be referred to by its symbolic name; these symbolic names are defined by the \$LKIDEF macro. The following lists, in order, the symbolic name of each bit field.

Symbolic Name	Description
LKISW_GROUP	<p>In this field (bits 0 to 15) \$GETLKI writes the UIC group number of the process that took out the first lock on the resource, thereby creating the resource name. This process issued a call to \$ENQ or \$ENQW specifying the name of the resource in the resnam argument.</p> <p>However, if this process specified the LCK\$_SYSTEM flag in the call to \$ENQ or \$ENQW, the resource name is system wide. In this case, the UIC group number of the process is not associated with the resource name.</p> <p>Consequently, this field (bits 0 to 15) is significant only if the resource name is not system wide. \$GETLKI sets bit 31 if the resource name is system wide.</p>
LKISB_RMOD	<p>In this field (bits 16 to 23) \$GETLKI writes the access mode associated with the first lock taken out on the resource.</p>
LKISB_STATUS	<p>This field (bits 24 to 30) is not used. \$GETLKI sets it to 0.</p>
LKISV_SYSNAM	<p>This field (bit 31) indicates whether the resource name is system wide. \$GETLKI sets this bit if the resource name is system wide and clears it if the resource name is qualified by the creating process's UIC group number. The state of this bit determines the interpretation of bits 0 to 15.</p>

LKIS_PARENT

When LKIS_PARENT is specified, \$GETLKI returns the lock id of the parent lock for the lock, if a parent lock was specified in the call to \$ENQ or \$ENQW. If the lock does not have a parent lock, \$GETLKI returns the value 0.

Since the parent lock id is a longword in length, the **buffer length** field in the item descriptor should specify 4 (bytes).

LKIS_PID

When LKIS_PID is specified, \$GETLKI returns the process identification (process id) of the process that owns the lock.

System Service Descriptions

\$GETLKI

The process id is a longword value, so the **buffer length** field in the item descriptor should specify 4 (bytes).

LKI\$_REMLKID

When LKI\$_REMLKID is specified, \$GETLKI returns the remote lock id, if one exists, for the current lock. The remote lock id is the lock id of the corresponding lock on another node in the cluster.

In a VAXcluster, each VAX node maintains its own lock data base. This data base lists the name of each resource, the process id of each process having a lock granted on this resource, the name of the VAX node on which each of these processes resides, the type of lock held by each process, and the lock id's of the locks held by each of these processes. However, these lock id's are node specific and are truly useful only when the lock identified by the lock id is held by a process on the local (the caller of \$GETLKI's) node. In such cases, the lock id absolutely identifies the lock and could, for example, be specified in a call to \$ENQ or \$DEQ in order to dequeue the lock, change its lock mode, and so on.

In cases where the lock id identifies a lock held by a process on another (remote) VAX node, the lock id as specified on the local node cannot be used to influence events on the remote node where the holder of the lock resides. The reason for this is that the remote node's lock data base uses a different lock id to identify that lock. However, by specifying the LKI\$_REMLKID item code, the caller of \$GETLKI can obtain this remote lock id and use it to manipulate the lock on the remote node.

The remote lock id is a longword value, so the **buffer length** field in the item descriptor should specify 4 (bytes).

LKI\$_RESNAM

When LKI\$_RESNAM is specified, \$GETLKI returns the resource name string and its length, which must be from 1 to 31 bytes. The resource name string was specified in the **resnam** argument in the initial call to \$ENQ or \$ENQW.

\$GETLKI returns the length of the string in the **return length address** field in the item descriptor. However, in the call to \$GETLKI, you will not know how long the string is. Therefore, to avoid buffer overflow, you should specify the maximum length (31 bytes) in the **buffer length** field in the item descriptor.

LKI\$_RSBREFCNT

When LKI\$_RSBREFCNT is specified, \$GETLKI returns the number of sub-resources of the resource associated with the lock. A sub-resource has the resource as a parent resource. Note however that the number of sub-resources may differ from the number of sub-locks of the lock.

This is true because any number of processes may lock the resource. If any of these processes then locks another resource, and in doing so specifies the lock id of the lock on the first resource as a parent lock, then the second resource becomes a sub-resource of the first resource.

Thus, the number of sub-locks on a lock is limited to the number of sub-locks that a single process takes out; whereas the number of sub-resources on a resource is determined by (potentially) multiple processes.

The sub-resource reference count is a longword value, so the **buffer length** field in the item descriptor should specify 4 (bytes).

System Service Descriptions

\$GETLKI

LKIS_STATE

When LKIS_STATE is specified, \$GETLKI returns the current state of the lock. The current state of the lock is described by the following three one-byte items (in the order specified): (1) the lock mode requested (in the call to \$ENQ or \$ENQW) for the lock, (2) the lock mode granted (by \$ENQ or \$ENQW) for the lock, and (3) the name of the queue on which the lock currently resides.

The **buffer length** field in the item descriptor should specify 3 (bytes). The three 1-byte items in the buffer may be referred to by the following symbolic names, which are defined by the \$LKIDEF macro:

Symbolic Name	Description
LKISB_STATE_RQMODE	Lock mode requested
LKISB_STATE_GRMODE	Lock mode granted
LKISB_STATE_QUEUE	Name of queue on which the lock resides

The values that \$GETLKI may write into each 1-byte item have symbolic names; these symbolic names specify the six lock modes and the three types of queue in which a lock may reside. The Description section lists these names.

LKIS_SYSTEM

When LKIS_SYSTEM is specified, \$GETLKI returns the cluster system identifier (CSID) of the VAX node that is maintaining the locked resource. Though the resource may be locked by processes on any node in the cluster, the resource itself is maintained on a single node.

You can use the DCL SHOW CLUSTER command or the \$GETSYI service to determine which VAX node in the cluster is identified by the CSID that \$GETLKI returns.

Since the CSID is a 4-byte string, the **buffer length** field in the item descriptor should specify 4.

This item code is useful only in VAXcluster environments.

LKIS_VALBLK

When LKIS_VALBLK is specified, \$GETLKI returns the lock value block of the locked resource. This lock value block is the master copy that the lock manager maintains for the resource, not the process-private copy.

As the lock value block is 16 bytes, the **buffer length** field in the item descriptor should specify 16.

iosb

VMS Usage: **io_status_block**

type: **quadword (unsigned)**

access: **write only**

mechanism: **by reference**

I/O Status Block that is to receive the final completion status. The **iosb** argument is the address of a quadword.

When \$GETLKI is called, it sets the I/O status block to 0. When \$GETLKI completes, it writes a condition value to the first longword in the quadword. The remaining two words in the quadword are unused. These condition values are listed after the Description section.

System Service Descriptions

\$GETLKI

Though this argument is optional, DIGITAL strongly recommends that you specify it for the following reasons.

- If you are using an event flag to signal the completion of the service, you can test the I/O status block for a condition value to be sure that the event flag was not set by an event other than service completion.
- If you are using the \$SYNCH service to synchronize completion of the service, the I/O status block is a required argument for \$SYNCH.
- The condition value returned in R0 and the condition value returned in the I/O status block provide information about different aspects of the call to the \$GETLKI service. The condition value returned in R0 gives you information about the success or failure of the service call itself; the condition value returned in the I/O status block gives you information about the success or failure of the service operation. Therefore, to accurately assess the success or failure of the call to \$GETLKI, you must check the condition values returned in both R0 and the I/O status block.

astadr

VMS Usage: **ast_procedure**
type: **procedure entry mask**
access: **call without stack unwinding**
mechanism: **by reference**

AST service routine to be executed when the service completes. The **astadr** argument is the address of the entry mask of this routine.

If this argument is specified, the AST routine will execute at the same access mode as the caller of the \$GETLKI service.

astprm

VMS Usage: **user_arg**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

AST parameter to be passed to the AST service routine specified by the **astadr** argument. The **astprm** argument is the longword parameter.

nullarg

VMS Usage: **null_arg**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Place-holding argument. This argument is reserved to DIGITAL.

DESCRIPTION

Depending on the operation, use of \$GETLKI may require the calling process to have certain privileges.

- WORLD privilege is required to obtain information about locks held by processes in other groups
- To obtain information about system locks, either SYSLOCK privilege is required or the process must be executing in executive or kernel access mode

System Service Descriptions

\$GETLKI

The access mode of the calling process must be equal to or more privileged than the access mode at which the lock was initially granted.

When locking on a resource is cluster-wide, a single master copy of the resource is maintained on the node that owns the process that created the resource by taking out the first lock on it. When a process on another VAX node locks that same resource, a local copy of the resource is copied to the node and the lock is identified by a lock id that is unique to that node.

However, in a VAXcluster environment you cannot use \$GETLKI to obtain directly information about locks on other nodes in the cluster; which is to say, you cannot specify in a call to \$GETLKI the lock id of a lock held by a process on another node. \$GETLKI interprets the **lkidadr** argument as the lock id of a lock on the caller's node, even though the resource associated with a lock may or may not have its master copy on the caller's node.

However, since a process on another node in the cluster may have a lock on the same resource as does the caller of \$GETLKI on this node, the caller on this node, in obtaining information about the resource, may indirectly obtain some information about locks on the resource that are held by processes on other nodes. One example of information indirectly obtained about a resource is the contents of lock queues; these queues contain information about all locks on the resource, and some of these locks may be held by processes on other nodes.

Another example of information more directly obtained is the remote lock id of a lock held by a process on another node. Specifically, if the caller of \$GETLKI on node A specifies a lock (via **lkidadr**) and that lock is held by a process on node B, \$GETLKI will return the lock id of the lock from node B's lock database if the **LKI\$_REMLKID** item code is specified in the call.

Item codes **LKI\$_BLOCKEDBY**, **LKI\$_BLOCKING**, **LKI\$_LOCKS**, and **LKI\$_STATE** specify that \$GETLKI return various items of information; some of these items are the names of lock modes or the names of lock queues. These names have symbolic names that are defined by the **\$LCKDEF** macro. The symbolic names are as follows:

Symbolic Name	Lock Mode
LCK\$_NLMODE	Null mode
LCK\$_CRMODE	Concurrent read mode
LCK\$_CWMODE	Concurrent write mode
LCK\$_PRMODE	Protected read mode
LCK\$_PWMODE	Protected write mode
LCK\$_EXMODE	Exclusive mode

Symbolic Name	Queue Name
LKISC\$_GRANTED	Granted queue, holding locks that have been granted
LKISC\$_CONVERT	Converting queue, holding locks that are currently being converted to another lock mode
LKISC\$_WAITING	Waiting queue, holding locks that are neither granted nor converting (for example, a blocked lock)

System Service Descriptions

\$GETLKI

CONDITION VALUES RETURNED

SS\$_NORMAL
SS\$_NOMORELOCK

Successful completion.

The caller requested a wildcard operation by specifying a value of 0 or -1 for the **lkidadr** argument, and \$GETLKI has exhausted the locks about which it can return information to the caller. This is an alternate success status.

SS\$_ACCVIO

The item list cannot be read; the areas specified by the **buffer address** and **return length address** fields in the item descriptor cannot be written.

SS\$_BADPARAM

An invalid item code was specified.

SS\$_IVLOCKID

The **lkidadr** argument specified an invalid lock id.

SS\$_IVMODE

A more privileged access mode is required.

SS\$_NOSYLCK

The caller attempted to acquire information about a system wide lock and did not have the required SYSLCK privilege.

SS\$_NOWORLD

The caller attempted to acquire information about a lock held by a process in another group and did not have the required WORLD privilege.

SS\$_EXQUOTA

The caller has insufficient ASTLM or BYTLM quota.

SS\$_INSFMEM

There is insufficient nonpaged dynamic memory for the operation.

CONDITION VALUES RETURNED IN THE I/O STATUS BLOCK

Same as those returned in R0.

System Service Descriptions

\$GETLKIW

\$GETLKIW—Get Lock Information and Wait

The Get Lock Information and Wait service returns information about the lock data base on a VMS system.

The \$GETLKIW service completes synchronously; that is, it returns to the caller with the requested information.

For asynchronous completion, use the Get Lock Information (\$GETLKI) service; \$GETLKI returns to the caller after queuing the information request, without waiting for the information to be returned.

In all other respects, \$GETLKIW is identical to \$GETLKI. Refer to the documentation of \$GETLKI for all other information about the \$GETLKIW service.

For additional information about system service completion, refer to the Synchronize (\$SYNCH) service and to Section 2.5.

The \$GETLKI, \$GETLKIW, \$ENQ (Enqueue Lock Request), \$ENQW (Enqueue Lock Request and Wait), and \$DEQ (Dequeue Lock Request) services together provide the user interface to the VAX/VMS lock management facility. Refer to the descriptions of these other services and to Section 12 in Part I for additional information about lock management.

FORMAT	SYSS\$GETLKIW <i>[efn] ,lkidadr ,itmlst [,iosb] [,astadr] [,astprm] [,nullarg]</i>
---------------	---

\$GETMSG—Get Message

The Get Message service locates and returns message text associated with a given message identification code into the caller's buffer. The message can be from the system message file or can be a user-defined message.

FORMAT

SY\$GETMSG *msgid,msglen,bufadr,[flags]*
,[outadr]

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS

msgid

VMS Usage: cond_value
type: longword (unsigned)
access: read only
mechanism: by value

Identification of the message to be retrieved. The **msgid** argument is a longword value containing the message identification. Each message has a unique identification, contained in bits 3 through 27 of system longword condition values.

msglen

VMS Usage: **word_unsigned**
type: **word (unsigned)**
access: **write only**
mechanism: **by reference**

Length of the message string returned by \$GETMSG. The **msglen** argument is the address of a word into which \$GETMSG writes this length.

bufadr

VMS Usage: **char_string**
type: **character-coded text string**
access: **write only**
mechanism: **by descriptor—fixed length**

Buffer to receive the message string. The **bufadr** argument is the address of a character string descriptor pointing to the buffer into which **\$GETMSG** writes the message string. The maximum size of any message string is 256 bytes.

System Service Descriptions

\$GETMSG

flags

VMS Usage: **mask_longword**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Message components to be returned. The **flags** argument is a longword bit vector, wherein a bit when set specifies that that message component is to be returned. The following list describes the significant bits:

Bit	Value	Description
0	1	Include text of message
	0	Do not include text of message
1	1	Include message identifier
	0	Do not include message identifier
2	1	Include severity indicator
	0	Do not include severity indicator
3	1	Include facility name
	0	Do not include facility name

If this argument is omitted in a VAX MACRO or BLISS-32 service call, it defaults to a value of 15; that is, all flags are set and all components of the message are returned. If this argument is omitted in a FORTRAN service call, it defaults to a value of 0; the value 0 causes \$GETMSG to use the process default flags.

outadr

VMS Usage: **vector_byte_unsigned**
type: **byte (unsigned)**
access: **write only**
mechanism: **by reference**

Optional information to be returned by \$GETMSG. The **outadr** is the address of a four-byte array into which \$GETMSG writes the following information:

Byte	Contents
0	Reserved
1	Count of FAO arguments associated with message
2	User-specified value in message, if any
3	Reserved

DESCRIPTION VAX/VMS uses this service to retrieve messages based on unique message identifications and to prepare to output the messages.

The message identifications correspond to the symbolic names for condition values returned by system components, for example, SS\$_code from system services, RMS\$_code for VAX RMS messages, and so on.

When all bits in the **flags** argument are set, \$GETMSG returns a string in the following format:

System Service Descriptions

\$GETMSG

facility-severity-ident, message-text

The following list describes the information provided by each segment of the \$GETMSG return string.

facility	Identifies the component of the operating system
severity	Is the severity code (the low-order three bits of the condition value)
ident	Is the unique message identifier
message-text	Is the text of the message

For example, if the following **msgid** argument is specified, the \$GETMSG service returns the following string:

msgid argument

MSGID=SS\$_DUPLNAM

\$GETMSG service return string

%SYSTEM-F-DUPLNAM, duplicate process name

This service does not check the length of the argument list and therefore cannot return the SS\$_INSFARG (insufficient arguments) condition value: therefore, be sure to specify the required number of arguments.

You can define your own messages with the MESSAGE utility. See the *VAX/VMS Utilities Reference Volume* for additional information.

The message text associated with a particular 32-bit message identification can be retrieved from one of several places. This service takes the following steps to locate the message text:

- 1 All message sections linked into the currently executing image are searched for the associated information.
- 2 If the information is not found, the process-permanent message file is searched. (The process-permanent message file can be specified by the SET MESSAGE command.)
- 3 If the information is not found, the system-wide message file is searched.
- 4 If the information is not found, the SS\$_MSGNOTFND condition value is returned in R0 and a message in the following form is returned to the caller's buffer.

**%facility-severity-NONAME, message=xxxxxxxx[hex],
(facility=n, message=n[dec])**

CONDITION VALUES RETURNED

SS\$_NORMAL	Service successfully completed.
SS\$_BUFFEROVF	Service successfully completed. The string returned overflowed the buffer provided and has been truncated.
SS\$_MSGNOTFND	Service successfully completed; however, the message code cannot be found, and a default message has been returned.

System Service Descriptions

\$GETMSG

EXAMPLE

```
CODE:  .LONG  SS$_DUPLNAM      ; message identification
LENGTH: .WORD  0
BUFDESC:
        .LONG  256
        .ADDRESS -
                BUFFER
BUFFER:  .BLKB  256
FLAGS:  .WORD  'B0001          ; message flags - text only
        .
        $GETMSG_S -
                MSGID=CODE, -
                MSGLEN=LENGTH, -
                BUFADR=BUFDESC, -
                FLAGS=FLAGS
```

The above example shows a segment of a program used to obtain only the text portion of the message associated with the system message code SS\$_DUPLNAM. \$GETMSG returns the following string:

duplicate process name.

\$GETQUI—Get Queue Information

The Get Queue Information (\$GETQUI) service returns information about queues and the jobs initiated from those queues. The \$GETQUI and \$SNDJBC services together provide the user interface to the VAX/VMS Job Controller, which is the VAX/VMS queue and accounting manager. See the "Description" section of the \$SNDJBC service for a discussion of the different types of jobs and queues.

The \$GETQUI service completes asynchronously; that is, it returns to the caller after queuing the request, without waiting for the operation to complete.

For synchronous completion, use the Get Queue Information and Wait (\$GETQUIW) service. The \$GETQUIW service is identical to \$GETQUI in every way except that \$GETQUIW returns to the caller after the operation has completed.

For additional information about system service completion, refer to the Synchronize (\$SYNCH) service and to Section 2.5.

FORMAT

SYSS\$GETQUI [*efn*] ,*func* [,*nullarg*] [,*itmlst*] [,*iosb*] [,*astadr*] [,*astprm*]

RETURNS

VMS Usage: **cond_value**
 type: **longword (unsigned)**
 access: **write only**
 mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS *efn*

VMS Usage: **ef_number**
 type: **longword (unsigned)**
 access: **read only**
 mechanism: **by value**

Number of the event flag to be set when \$GETQUI completes. The *efn* argument is a longword containing this number. The *efn* argument is optional.

When the request is queued, \$GETQUI clears the specified event flag (or event flag 0 if *efn* was not specified). Then when the operation has completed, \$GETQUI sets the specified event flag (or event flag 0).

func

VMS Usage: **function_code**
 type: **word (unsigned)**
 access: **read only**

System Service Descriptions

\$GETQUI

mechanism: **by value**

Function code specifying the function that \$GETQUI is to perform. The **func** argument is a word containing this function code. The \$QUIDEF macro defines the names of each function code.

Only one function code may be specified in a single call to \$GETQUI. Most function codes require or allow for additional information to be passed in the call. This information is passed by using the **itmlst** argument, which specifies a list of one or more item descriptors. Each item descriptor in turn specifies an item code, which either describes the specific information to be returned by \$GETQUI, or otherwise affects the action designated by the function code.

The following list specifies each function code, describes the action it designates, and lists which item code(s) are applicable; descriptions of the item codes appear in the description of the **itmlst** argument.

\$GETQUI Function Codes with Their Valid Item Codes

QUI\$_CANCEL_OPERATION

This request terminates any wildcard operation that may have been initiated by a previous call to \$GETQUI by releasing the GETQUI context block (GQC) that the system maintains for your process. Because only one wildcard search sequence can be outstanding at any one time, you do not have to specify any item codes.

When you call \$GETQUI to perform a series of wildcard requests to retrieve information about characteristics, forms, or queues (and their associated jobs and files), the job controller maintains a GQC between calls that points to the next object in the wildcard sequence. The system retains this information until: (1) you have made calls to \$GETQUI to examine every object in the sequence; (2) your process has terminated; or (3) you explicitly cancel the wildcard operation by using the QUI\$_CANCEL_OPERATION function code. If your calls to \$GETQUI have located all the objects in the sequence in which you are interested, you should terminate the wildcard operation. This frees job controller resources and allows you to initiate another \$GETQUI operation.

QUI\$_DISPLAY_CHARACTERISTIC

This request returns information about a specific characteristic definition, or the next characteristic definition in a wildcard operation.

A successful QUI\$_DISPLAY_CHARACTERISTIC wildcard operation terminates when the \$GETQUI service has returned information about all characteristic definitions included in the wildcard sequence. The \$GETQUI service signals termination of this sequence by returning the condition value JBC\$_NOMORECHAR in the I/O status block. If the \$GETQUI service does not find any characteristic definitions, it returns the condition value JBC\$_NOSUCHCHAR in the I/O status block.

For more information on how to request information about characteristics, see the "Description" section.

One of the following input item codes must be specified. Both may be specified.

QUI\$_SEARCH_NAME
QUI\$_SEARCH_NUMBER

The following input item code may be specified.

QUI\$_SEARCH_FLAGS

System Service Descriptions

\$GETQUI

The following output item codes may be specified.

QUI\$_CHARACTERISTIC_NAME
QUI\$_CHARACTERISTIC_NUMBER

QUI\$_DISPLAY_FILE

This request is normally made as part of a wildcard queue-job-file sequence of operations; that is, before you make a call to \$GETQUI to request file information, you have already made two previous calls to the \$GETQUI service to establish the queue and job context of the queue and job that contain the files in which you are interested. In this case, the \$GETQUI service returns information about the next file defined for the current job context. The \$GETQUI service signals that it has returned information about all the files defined for the current job context by returning the condition value JBC\$_NOMOREFILE in the I/O status block. If the current job context contains no files, \$GETQUI returns the condition value JBC\$_NOSUCHFILE in the I/O status block.

A batch job can make a call to the \$GETQUI service to request information about the command file that is currently executing without first making calls to the service to establish a queue and job context. To do this, the batch job specifies the QUI\$_V_SEARCH_THIS_JOB option of the QUI\$_SEARCH_FLAGS item code. The system does not save the queue or job context established in such a call.

For more information on how to request file information, see the "Description" section.

The following input item code may be specified.

QUI\$_SEARCH_FLAGS

The following output item codes may be specified.

QUI\$_FILE_COPIES
QUI\$_FILE_COPIES_DONE
QUI\$_FILE_FLAGS
QUI\$_FILE_SETUP_MODULES
QUI\$_FILE_SPECIFICATION
QUI\$_FILE_STATUS
QUI\$_FIRST_PAGE
QUI\$_LAST_PAGE

QUI\$_DISPLAY_FORM

This request returns information about a specific form definition, or the next form definition in a wildcard operation.

A successful QUI\$_DISPLAY_FORM wildcard operation terminates when the \$GETQUI service has returned information about all form definitions included in the wildcard sequence. The \$GETQUI service signals termination of this wildcard sequence by returning the condition value JBC\$_NOMOREFORM in the I/O status block. If the \$GETQUI service does not find any form definitions, it returns the condition value JBC\$_NOSUCHFORM in the I/O status block.

For more information on how to request information about forms, see the "Description" section.

One of the following input item codes must be specified. Both may be specified.

QUI\$_SEARCH_NAME

System Service Descriptions

\$GETQUI

QUI\$_SEARCH_NUMBER

The following input item code may be specified.

QUI\$_SEARCH_FLAGS

The following output item codes may be specified.

QUI\$_FORM_DESCRIPTION
QUI\$_FORM_FLAGS
QUI\$_FORM_LENGTH
QUI\$_FORM_MARGIN_BOTTOM
QUI\$_FORM_MARGIN_LEFT
QUI\$_FORM_MARGIN_RIGHT
QUI\$_FORM_MARGIN_TOP
QUI\$_FORM_NAME
QUI\$_FORM_NUMBER
QUI\$_FORM_SETUP_MODULES
QUI\$_FORM_STOCK
QUI\$_FORM_WIDTH
QUI\$_PAGE_SETUP_MODULES

QUI\$_DISPLAY_JOB

This request is normally made as part of a wildcard queue-job sequence of operations; that is, before you make a call to \$GETQUI to request job information, you have already made a call to the \$GETQUI service to establish the queue context of the queue that contains the job in which you are interested. In this case, the \$GETQUI service returns information about the next job defined for the current queue context.

The QUI\$_DISPLAY_JOB operation also establishes a job context for a subsequent QUI\$_DISPLAY_FILE operation. The job context established remains in effect until another call is made to the \$GETQUI service that specifies the QUI\$_DISPLAY_JOB, QUI\$_DISPLAY_QUEUE, or QUI\$_CANCEL_OPERATION function code.

The \$GETQUI service signals that it has returned information about all the jobs contained in the current queue context by returning the condition value JBC\$_NOMOREFILE in the I/O status block. If the current queue context contains no jobs, \$GETQUI returns the condition value JBC\$_NOSUCHFILE in the I/O status block.

A batch job can make a call to the \$GETQUI service to request information about itself without first making a call to the service to establish a queue context. To do this, the batch job must specify the QUI\$V_SEARCH_THIS_JOB option of the QUI\$_SEARCH_FLAGS item code. The system does not save the queue or job context established in such a call.

For more information on how to request job information, see the "Description" section.

The following input item code may be specified.

QUI\$_SEARCH_FLAGS

The following output item codes may be specified.

QUI\$_ACCOUNT_NAME
QUI\$_AFTER_TIME
QUI\$_CHARACTERISTICS
QUI\$_CHECKPOINT_DATA
QUI\$_CLI

System Service Descriptions

\$GETQUI

QUI\$_COMPLETED_BLOCKS
QUI\$_CONDITION_VECTOR
QUI\$_CPU_LIMIT
QUI\$_ENTRY_NUMBER
QUI\$_FORM_NAME
QUI\$_INTERVENING_BLOCKS
QUI\$_INTERVENING_JOBS
QUI\$_JOB_COPIES
QUI\$_JOB_COPIES_DONE
QUI\$_JOB_FLAGS
QUI\$_JOB_NAME
QUI\$_JOB_PID
QUI\$_JOB_SIZE
QUI\$_JOB_STATUS
QUI\$_LOG_QUEUE
QUI\$_LOG_SPECIFICATION
QUI\$_NOTE
QUI\$_OPERATOR_REQUEST
QUI\$_PARAMETER_1 through 8
QUI\$_PRIORITY
QUI\$_QUEUE_NAME
QUI\$_QUEUE_QUEUE_NAME
QUI\$_SUBMISSION_TIME
QUI\$_UIC
QUI\$_USERNAME
QUI\$_WSDEFAULT
QUI\$_WSEXTENT
QUI\$_WSQUOTA

QUI\$_DISPLAY_QUEUE

This request returns information about a specific queue definition, or the next queue definition in a wildcard operation.

If the call is successful, the QUI\$_DISPLAY_QUEUE operation also establishes a queue context for subsequent QUI\$_DISPLAY_JOB operations. The queue context established remains in effect until another call is made to the \$GETQUI service that specifies either the QUI\$_DISPLAY_QUEUE or QUI\$_CANCEL_OPERATION function code.

The \$GETQUI service signals that it has returned information about all the queues contained in the current wildcard sequence by returning the condition value JBC\$_NOMOREQUE in the I/O status block. If no queue is found, \$GETQUI returns the condition value JBC\$_NOSUCHQUE in the I/O status block.

A batch job can make a call to the \$GETQUI service to request information about the queue in which it is contained without first making a call to the service to establish a queue context. To do this, the batch job must specify the QUI\$V_SEARCH_THIS_JOB option of the QUI\$_SEARCH_FLAGS item code. The system does not save the queue context established in such a call.

For more information on how to request queue information, see the "Description" section.

The following input item code must be specified.

QUI\$_SEARCH_NAME

System Service Descriptions

\$GETQUI

The following input item code may be specified.

QUI\$_SEARCH_FLAGS

The following output item codes may be specified.

QUI\$_ASSIGNED_QUEUE_NAME
QUI\$_BASE_PRIORITY
QUI\$_CHARACTERISTICS
QUI\$_CPU_DEFAULT
QUI\$_CPU_LIMIT
QUI\$_DEFAULT_FORM_NAME
QUI\$_DEFAULT_FORM_STOCK
QUI\$_DEVICE_NAME
QUI\$_FORM_NAME
QUI\$_GENERIC_TARGET
QUI\$_JOB_LIMIT
QUI\$_JOB_RESET_MODULES
QUI\$_JOB_SIZE_MAXIMUM
QUI\$_JOB_SIZE_MINIMUM
QUI\$_LIBRARY_SPECIFICATION
QUI\$_OWNER_UIC
QUI\$_PROCESSOR
QUI\$_PROTECTION
QUI\$_QUEUE_FLAGS
QUI\$_QUEUE_NAME
QUI\$_QUEUE_STATUS
QUI\$_SCSNODE_NAME
QUI\$_WSDEFAULT
QUI\$_WSEXTENT
QUI\$_WSQUOTA

QUI\$_TRANSLATE_QUEUE

This request translates a logical name for a queue to the equivalence name for the queue. The logical name is specified by QUI\$_SEARCH_NAME. The translation is performed iteratively until the equivalence string is found or the number of translations allowed by the system has been reached.

The following input item code must be specified.

QUI\$_SEARCH_NAME

The following output item code may be specified.

QUI\$_QUEUE_NAME

nullarg

VMS Usage: **null_arg**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Place-holding argument. This argument is reserved to DIGITAL.

itmlst

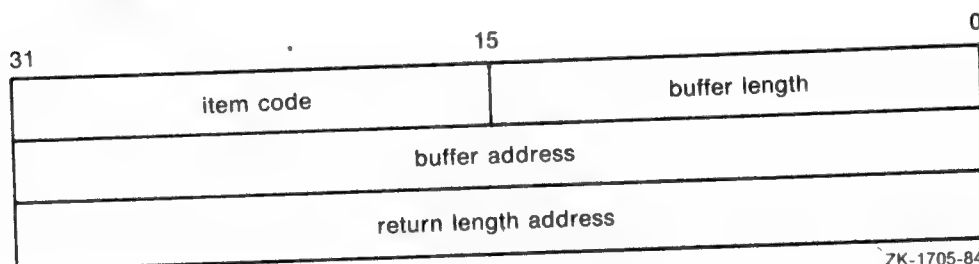
VMS Usage: **item_list_3**
type: **longword (unsigned)**
access: **read only**

System Service Descriptions

\$GETQUI

mechanism: **by reference**

Item list supplying information to be used in performing the function specified by the **func** argument. The **itmlst** argument is the address of the item list. The item list consists of one or more item descriptors, each of which specifies an item code. The item list is terminated by an item code of 0 or by a longword of 0. The following diagram depicts the structure of a single item descriptor.



ZK-1705-84

\$GETQUI Item Descriptor Fields

buffer length

A word specifying the length of the buffer; the buffer either supplies information to be used by \$GETQUI or receives information from \$GETQUI. The required length of the buffer varies depending on the item code specified and is given in the description of each item code.

item code

A word containing an item code, which identifies the nature of the information that is supplied for use by \$GETQUI or that is received from \$GETQUI. Each item code has a symbolic name; these symbolic names are defined by the \$QUIDEF macro and have the format: QUI\$_code.

There are two types of item code:

- Input value item code. \$GETQUI has only three input value item codes: QUI\$_SEARCH_NAME, QUI\$_SEARCH_NUMBER and QUI\$_SEARCH_FLAGS. These item codes specify the object name or number for which \$GETQUI is to return information and specify the extent of \$GETQUI's search for these objects. Most function codes require that you specify at least one input value item code. The required and optional input value item codes for each function code are listed following the function code description.

For input value item codes, the **buffer length** and **buffer address** fields of the item descriptor must be nonzero; the return length field must be zero. Specific buffer length requirements are given in the description of each item code.

- Output value item code. Output value item codes specify a buffer for information returned by \$GETQUI. For output value item codes, the **buffer length** and **buffer address** fields of the item descriptor must be nonzero; the return length field may be zero or nonzero. Specific buffer length requirements are given in the description of each item code.

System Service Descriptions

\$GETQUI

Several item codes specify a queue name, form name, or characteristic name to \$GETQUI, or request that \$GETQUI return one of these names. For these item codes, the buffer must specify or be prepared to receive a string containing from 1 to 31 characters, exclusive of spaces, tabs, and null characters, which are ignored. Allowable characters in the string are the uppercase alphabetic characters, the lowercase alphabetic characters (which are converted to uppercase), the numeric characters, the dollar sign (\$), and the underscore (_).

buffer address

Address of the buffer that specifies or receives the information.

return length address

Address of a word to receive the length of information returned by \$GETQUI.

\$GETQUI Item Codes

QUI\$_ACCOUNT_NAME

When QUI\$_ACCOUNT_NAME is specified, \$GETQUI returns the 8-byte account name of the owner of the specified job.

(QUI\$_DISPLAY_JOB item code)

QUI\$_AFTER_TIME

When QUI\$_AFTER_TIME is specified, \$GETQUI returns, as a quadword absolute time value, the system time at or after which the specified job can execute.

(QUI\$_DISPLAY_JOB item code)

QUI\$_ASSIGNED_QUEUE_NAME

When QUI\$_ASSIGNED_QUEUE_NAME is specified, \$GETQUI returns, as a character string, the name of the execution queue to which the logical queue specified in the call to \$GETQUI is assigned. Since the queue name can include up to 31 characters, the buffer length field of the item descriptor should specify 31 (bytes).

(QUI\$_DISPLAY_QUEUE item code)

QUI\$_BASE_PRIORITY

When QUI\$_BASE_PRIORITY is specified, \$GETQUI returns, as a longword value in the range 0 to 15, the priority at which batch jobs are initiated from a batch execution queue or the priority of a symbiont process that controls output execution queues.

(QUI\$_DISPLAY_QUEUE item code)

QUI\$_CHARACTERISTIC_NAME

When QUI\$_CHARACTERISTIC_NAME is specified, \$GETQUI returns, as a character string, the name of the specified characteristic. Since the characteristic name can include up to 31 characters, the buffer length field of the item descriptor should specify 31 (bytes).

(QUI\$_DISPLAY_CHARACTERISTIC item code)

QUI\$_CHARACTERISTIC_NUMBER

When QUI\$_CHARACTERISTIC_NUMBER is specified, \$GETQUI returns, as a longword value in the range 0 to 127, the number of the specified characteristic.

System Service Descriptions

\$GETQUI

(QUI\$_DISPLAY_CHARACTERISTIC item code)

QUI\$_CHARACTERISTICS

When QUI\$_CHARACTERISTICS is specified, \$GETQUI returns, as a 128-bit string (16-byte field), the characteristics associated with the specified queue or job. Each bit set in the bitmask represents a characteristic number in the range 0 to 127.

(QUI\$_DISPLAY_JOB, QUI\$_DISPLAY_QUEUE item code)

QUI\$_CHECKPOINT_DATA

When QUI\$_CHECKPOINT_DATA is specified, \$GETQUI returns, as a character string, the value of the DCL symbol BATCH\$RESTART when the specified batch job is restarted. Since the value of the symbol can include up to 255 characters, the buffer length field of the item descriptor should specify 255 (bytes).

(QUI\$_DISPLAY_JOB item code)

QUI\$_CLI

When QUI\$_CLI is specified, \$GETQUI returns, as an RMS file name component, the name of the command language interpreter used to execute the specified batch job. The file specification returned assumes the device name SYS\$SYSTEM: and the file type EXE. Since a file name can include up to 39 characters, the buffer length field in the item descriptor should specify 39 (bytes). This item code is applicable only to batch jobs.

(QUI\$_DISPLAY_JOB item code)

QUI\$_COMPLETED_BLOCKS

When QUI\$_COMPLETED_BLOCKS is specified, \$GETQUI returns, as a longword decimal value, the number of blocks that the symbiont has processed for the specified print job. This item code is applicable only to print jobs.

(QUI\$_DISPLAY_JOB item code)

QUI\$_CONDITION_VECTOR

When QUI\$_CONDITION_VECTOR is specified, \$GETQUI returns, as a longword condition value, the completion status of the specified job.

(QUI\$_DISPLAY_JOB item code)

QUI\$_CPU_DEFAULT

When QUI\$_CPU_DEFAULT is specified, \$GETQUI returns, as a longword decimal value, the default CPU time limit specified for the queue in 10-millisecond units. This item code is applicable only to batch execution queues.

(QUI\$_DISPLAY_QUEUE item code)

QUI\$_CPU_LIMIT

When QUI\$_CPU_LIMIT is specified, \$GETQUI returns, as a longword decimal value, the maximum CPU time limit specified for the specified job queue in 10-millisecond units. This item code is applicable only to batch and batch execution queues.

(QUI\$_DISPLAY_JOB, DISPLAY_QUEUE item code)

System Service Descriptions

\$GETQUI

QUI\$_DEFAULT_FORM_NAME

When QUI\$_DEFAULT_FORM_NAME is specified, \$GETQUI returns, as a character string, the name of the default form associated with the specified output queue. Since the form name can include up to 31 characters, the buffer length field of the item descriptor should specify 31 (bytes).

For more information on default forms, see Chapter 9 of the VAX/VMS System Manager's Reference Manual.

(QUI\$_DISPLAY_QUEUE item code)

QUI\$_DEFAULT_FORM_STOCK

When QUI\$_DEFAULT_FORM_STOCK is specified, \$GETQUI returns, as a character string, the name of the paper stock on which the specified default form is to be printed. Since the name of the paper stock can include up to 31 characters, the buffer length field of the item descriptor should specify 31 (bytes).

For more information on default forms, see Chapter 9 of the VAX/VMS System Manager's Reference Manual.

(QUI\$_DISPLAY_QUEUE item code)

QUI\$_DEVICE_NAME

When QUI\$_DEVICE_NAME is specified, \$GETQUI returns, as a 1 to 31 character string, the node and/or device on which the specified execution queue is located. For batch execution queues, only the node name is returned. For output execution queues, both the node name and the device name are returned. The node name is used only in VAXcluster systems. The node name is specified by the SYSGEN parameter SCSNODE for the processor on which the queue executes.

(QUI\$_DISPLAY_QUEUE item code)

QUI\$_ENTRY_NUMBER

When QUI\$_ENTRY_NUMBER is specified, \$GETQUI returns, as a longword decimal value, the queue entry number of the specified job.

(QUI\$_DISPLAY_JOB item code)

QUI\$_FILE_COPIES

When QUI\$_FILE_COPIES is specified, \$GETQUI returns the number of times the specified file is to be processed, which is a longword decimal value in the range 1 to 255. This item code is applicable only to output execution queues.

(QUI\$_DISPLAY_FILE item code)

QUI\$_FILE_COPIES_DONE

When QUI\$_FILE_COPIES_DONE is specified, \$GETQUI returns the number of times the specified file has been processed, which is a longword decimal value in the range 1 to 255. This item code is applicable only to output execution queues.

(QUI\$_DISPLAY_FILE item code)

QUI\$_FILE_FLAGS

When QUI\$_FILE_FLAGS is specified, \$GETQUI returns, as a longword bit vector, the processing options that have been selected for the specified file. Each processing option is represented by a bit. When \$GETQUI sets a bit, the file is processed according to the corresponding processing option. Each bit

System Service Descriptions

\$GETQUI

in the vector has a symbolic name. These symbolic names are defined by the \$QUIDEF macro and are listed below.

Symbol	Description
QUI\$V_FILE_BURST	Burst and flag pages are to be printed preceding a file.
QUI\$V_FILE_DELETE	File is to be deleted after execution of request.
QUI\$V_FILE_DOUBLE_SPACE	Symbiont formats the file with double spacing.
QUI\$V_FLAG	Flag page is to be printed preceding a file.
QUI\$V_FILE_TRAILER	Trailer page is to be printed following a file.
QUI\$V_FILE_PAGE_HEADER	Page header is to be printed on each page of output.
QUI\$V_FILE_PAGINATE	Symbiont paginates output by inserting a form feed whenever output reaches the bottom margin of the form.
QUI\$V_FILE_PASSALL	Symbiont prints the file in PASSALL mode.

(QUI\$_DISPLAY_FILE item code)

QUI\$_FILE_SETUP_MODULES

When QUI\$_FILE_SETUP_MODULES is specified, \$GETQUI returns, as a comma-separated list, the names of the text modules that are to be extracted from the device control library and copied to the printer before the specified file is printed. Since a text module name can include up to 31 characters and is separated from the previous text module name with a comma, the buffer length field of the item descriptor should specify 32 (bytes) for each possible text module. This item code is meaningful only for output execution queues.

(QUI\$_DISPLAY_FILE item code)

QUI\$_FILE_SPECIFICATION

When QUI\$_FILE_SPECIFICATION is specified, \$GETQUI returns the fully qualified RMS file specification of the file about which \$GETQUI is returning information. Since a file specification can include up to 255 characters, the buffer length field of the item descriptor should specify 255 (bytes) for each possible text module.

(QUI\$_DISPLAY_FILE item code)

QUI\$_FILE_STATUS

When QUI\$_FILE_STATUS is specified, \$GETQUI returns file status information as a longword bit vector. Each file status condition is represented by a bit. When \$GETQUI sets the bit, the file status corresponds to the condition represented by the bit. Each bit in the vector has a symbolic name. These symbolic names are defined by the \$QUIDEF macro and are listed below.

Symbol	Description
QUI\$V_FILE_CHECKPOINTED	File is checkpointed.
QUI\$V_FILE_EXECUTING	File is being processed.

(QUI\$_DISPLAY_FILE item code)

System Service Descriptions

\$GETQUI

QUI\$_FIRST_PAGE

When QUI\$_FIRST_PAGE is specified, \$GETQUI returns, as a longword decimal value, the page number at which the printing of the specified file is to begin. This item code is applicable only to output execution queues.

(QUI\$_DISPLAY_FILE item code)

QUI\$_FORM_DESCRIPTION

When QUI\$_FORM_DESCRIPTION is specified, \$GETQUI returns, as a character string, the text string that describes the specified form to users and operators. Since the text string can include up to 255 characters, the buffer length field in the item descriptor should specify 255 (bytes).

(QUI\$_DISPLAY_FORM item code)

QUI\$_FORM_FLAGS

When QUI\$_FORM_FLAGS is specified, \$GETQUI returns, as a longword bit vector, the processing options that have been selected for the specified form. Each processing option is represented by a bit. When \$GETQUI sets a bit, the form is processed according to the corresponding processing option. Each bit in the vector has a symbolic name. These symbolic names are defined by the \$QUIDEF macro and are listed below:

Symbol	Description
QUI\$_V_FORM_SHEET_FEED	Symbiont pauses at the end of each physical page so that another sheet of paper can be inserted.
QUI\$_V_FORM_TRUNCATE	Printer discards any characters that exceed the specified right margin.
QUI\$_V_FORM_WRAP	Printer prints any characters that exceed the specified right margin on the following line.

(QUI\$_DISPLAY_FORM item code)

QUI\$_FORM_LENGTH

When QUI\$_FORM_LENGTH is specified, \$GETQUI returns, as a longword decimal value, the physical length of the specified form in lines. This item code is applicable only to output execution queues.

(QUI\$_DISPLAY_FORM item code)

QUI\$_FORM_MARGIN_BOTTOM

When QUI\$_FORM_MARGIN_BOTTOM is specified, \$GETQUI returns, as a longword decimal value, the bottom margin of the specified form in lines.

(QUI\$_DISPLAY_FORM item code)

QUI\$_FORM_MARGIN_LEFT

When QUI\$_FORM_MARGIN_LEFT is specified, \$GETQUI returns, as a longword decimal value, the left margin of the specified form in characters.

(QUI\$_DISPLAY_FORM item code)

QUI\$_FORM_MARGIN_RIGHT

When QUI\$_FORM_MARGIN_RIGHT is specified, \$GETQUI returns, as a longword decimal value, the right margin of the specified form in characters.

(QUI\$_DISPLAY_FORM item code)

QUI\$_FORM_MARGIN_TOP

When QUI\$_FORM_MARGIN_TOP is specified, \$GETQUI returns, as a longword decimal value, the top margin of the specified form in lines.

(QUI\$_DISPLAY_FORM item code)

QUI\$_FORM_NAME

When QUI\$_FORM_NAME is specified, \$GETQUI returns, as a character string, the name of the specified form or the mounted form associated with the specified job or queue. Since the form name can include up to 31 characters, the buffer length field of the item descriptor should specify 31 (bytes).

For more information on mounted forms, see Chapter 9 of the *VAX/VMS System Manager's Reference Manual*.

(QUI\$_DISPLAY_FORM, QUI\$_DISPLAY_JOB, QUI\$_DISPLAY_QUEUE item codes)

QUI\$_FORM_NUMBER

When QUI\$_FORM_NUMBER is specified, \$GETQUI returns, as a longword decimal value, the number of the specified form.

(QUI\$_DISPLAY_FORM item code)

QUI\$_FORM_SETUP_MODULES

When QUI\$_FORM_SETUP_MODULES is specified, \$GETQUI returns, as a comma-separated list, the names of the text modules that are to be extracted from the device control library and copied to the printer before a file is printed on the specified form. Since a text module name can include up to 31 characters and is separated from the previous text module name by a comma, the buffer length field of the item descriptor should specify 32 (bytes) for each possible text module. This item code is meaningful only for output execution queues.

(QUI\$_DISPLAY_FORM item code)

QUI\$_FORM_STOCK

When QUI\$_FORM_STOCK is specified, \$GETQUI returns, as a character string, the name of the paper stock on which the specified form is to be printed. Since the name of the paper stock can include up to 31 characters, the buffer length field of the item descriptor should specify 31 (bytes).

For more information on forms, see Chapter 9 of the *VAX/VMS System Manager's Reference Manual*.

(QUI\$_DISPLAY_FORM item code)

QUI\$_FORM_WIDTH

When QUI\$_FORM_WIDTH is specified, \$GETQUI returns, as a longword decimal value, the width of the specified form in characters.

(QUI\$_DISPLAY_FORM item code)

QUI\$_GENERIC_TARGET

When QUI\$_GENERIC_TARGET is specified, \$GETQUI returns, as a comma-separated list, the names of the execution queues that are enabled to accept work from the specified generic queue. Since a queue name can include up to 31 characters and is separated from the previous queue name with a comma, the buffer length field of the item descriptor should specify 32 (bytes) for each possible queue name. A generic queue can send work to

System Service Descriptions

\$GETQUI

up to 124 execution queues. This item code is meaningful only for generic queues.

(QUI\$_DISPLAY_QUEUE item code)

QUI\$_INTERVENING_BLOCKS

When QUI\$_INTERVENING_BLOCKS is specified, \$GETQUI returns, as a longword decimal value, the number of blocks that are to be processed before the specified job can begin to execute. This item code is meaningful only for output execution queues.

(QUI\$_DISPLAY_JOB item code)

QUI\$_INTERVENING_JOBS

When QUI\$_INTERVENING_JOBS is specified, \$GETQUI returns, as a longword decimal value, the number of jobs that are to be processed before the specified job can begin to execute. This item code is meaningful only for output execution queues.

(QUI\$_DISPLAY_JOB item code)

QUI\$_JOB_COPIES

When QUI\$_JOB_COPIES is specified, \$GETQUI returns, as a longword decimal value, the number of times that the specified print job is to be repeated.

(QUI\$_DISPLAY_JOB item code)

QUI\$_JOB_COPIES_DONE

When QUI\$_JOB_COPIES_DONE is specified, \$GETQUI returns, as a longword decimal value, the number of times that the specified print job has been repeated.

(QUI\$_DISPLAY_JOB item code)

QUI\$_JOB_FLAGS

When QUI\$_JOB_FLAGS is specified, \$GETQUI returns, as a longword bit vector, the processing options that have been selected for the specified job. Each processing option is represented by a bit. When \$GETQUI sets a bit, the job is processed according to the corresponding processing option. Each bit in the vector has a symbolic name. These symbolic names are defined by the \$QUIDEF macro and are listed below.

Symbol	Description
QUI\$V_JOB_CPU_LIMIT	CPU time limit is specified for the job.
QUI\$V_JOB_FILE_BURST	Burst and flag pages precede each file in the job.
QUI\$V_JOB_FILE_BURST_ONE	Burst and flag pages precede only the first copy of the first file in the job.
QUI\$V_JOB_FILE_FLAG	Flag page precedes each file in the job.
QUI\$V_JOB_FILE_FLAG_ONE	Flag page precedes only the first copy of the first file in the job.
QUI\$V_JOB_FILE_PAGINATE	Symbiont paginates output by inserting a form feed whenever output reaches the bottom margin of the form.

System Service Descriptions

\$GETQUI

Symbol	Description
QUI\$V_JOB_FILE_TRAILER	Trailer page follows each file in the job.
QUI\$V_JOB_FILE_TRAILER_ONE	Trailer page follows only the last copy of the last file in the job.
QUI\$V_JOB_LOG_DELETE	Log file is deleted after it is printed.
QUI\$V_JOB_LOG_NULL	No log file is created.
QUI\$V_JOB_LOG_SPOOL	Job log file is queued for printing when job is complete.
QUI\$V_JOB_LOWERCASE	Job is to be printed on printer that can print both uppercase and lowercase letters.
QUI\$V_JOB_NOTIFY	Message is broadcast to terminal when job completes or aborts.
QUI\$V_JOB_RESTART	Job will restart after a system failure or can be requeued during execution.
QUI\$V_JOB_WSDEFAULT	Default working set size is specified for the job.
QUI\$V_JOB_WSEXTENT	Working set extent is specified for the job.
QUI\$V_JOB_WSQUOTA	Working set quota is specified for the job.

(QUI\$_DISPLAY_JOB item code)

QUI\$_JOB_LIMIT

When QUI\$_JOB_LIMIT is specified, \$GETQUI returns the number of jobs that can execute simultaneously on the specified queue, which is a longword decimal value in the range 1 to 255. This item code is applicable only to batch execution queues.

(QUI\$_DISPLAY_QUEUE item code)

QUI\$_JOB_NAME

When QUI\$_JOB_NAME is specified, \$GETQUI returns, as a character string, the name of the specified job. Since the job name can include up to 39 characters, the buffer length field of the item descriptor should specify 39 (bytes).

(QUI\$_DISPLAY_JOB item code)

QUI\$_JOB_PID

When QUI\$_JOB_PID is specified, \$GETQUI returns the process identification (PID) of the executing batch job in standard longword format.

(QUI\$_DISPLAY_JOB item code)

QUI\$_JOB_RESET_MODULES

When QUI\$_JOB_RESET_MODULES is specified, \$GETQUI returns, as a comma-separated list, the names of the text modules that are to be extracted from the device control library and copied to the printer before each job in the specified queue is printed. Since a text module name can include up to 31 characters and is separated from the previous text module name by a comma, the buffer length field of the item descriptor should specify 32 (bytes) for each possible text module. This item code is meaningful only for output execution queues.

(QUI\$_DISPLAY_QUEUE item code)

System Service Descriptions

\$GETQUI

QUI\$_JOB_SIZE

When QUI\$_JOB_SIZE is specified, \$GETQUI returns, as a longword decimal value, the total number of blocks in the specified print job.

(QUI\$_DISPLAY_JOB item code)

QUI\$_JOB_SIZE_MAXIMUM

When QUI\$_JOB_SIZE_MAXIMUM is specified, \$GETQUI returns, as a longword decimal value, the maximum number of blocks that a print job initiated from the specified queue can contain. This item code is applicable only to output execution queues.

(QUI\$_DISPLAY_QUEUE item code)

QUI\$_JOB_SIZE_MINIMUM

When QUI\$_JOB_SIZE_MINIMUM is specified, \$GETQUI returns, as a longword decimal value, the minimum number of blocks that a print job initiated from the specified queue can contain. This item code is applicable only to output execution queues.

(QUI\$_DISPLAY_QUEUE item code)

QUI\$_JOB_STATUS

When QUI\$_JOB_STATUS is specified, \$GETQUI returns the specified job's status flags, which are contained in a longword bit vector. Listed below are the symbolic names for these flags, which are defined by the \$QUIDEF macro.

Symbol	Description
QUI\$V_JOB_ABORTING	System is attempting to abort execution of job.
QUI\$V_JOB_EXECUTING	Job is executing or printing.
QUI\$V_JOB_HOLDING	Job will be held until it is explicitly released.
QUI\$V_JOB_INACCESSIBLE	Caller does not have READ access to the specific job and file information in the system queue file. Therefore, the QUI\$_DISPLAY_JOB and QUI\$_DISPLAY_FILE operations can return information for only the following output value item codes. QUI\$_AFTER_TIME QUI\$_COMPLETED_BLOCKS QUI\$_ENTRY_NUMBER QUI\$_INTERVENING_BLOCKS QUI\$_INTERVENING_JOBS QUI\$_JOB_SIZE QUI\$_JOB_STATUS.
QUI\$V_JOB_REFUSED	Job was refused by symbiont and is waiting for symbiont to accept it for processing.
QUI\$V_JOB_RETAINED	Job has been completed, but it is being retained in the queue.

System Service Descriptions

\$GETQUI

Symbol	Description
QUI\$V_JOB_STARTING	Job controller is starting to process the job and has begun communicating with an output symbiont or a job controller on another node.
QUI\$V_JOB_TIMED	Job is waiting for specified time to execute.

(QUI\$_DISPLAY_JOB item code)

QUI\$_LAST_PAGE

When QUI\$_LAST_PAGE is specified, \$GETQUI returns, as a longword decimal value, the page number at which the printing of the specified file should end. This item code is applicable only to output execution queues.

(QUI\$_DISPLAY_FILE item code)

QUI\$_LIBRARY_SPECIFICATION

When QUI\$_LIBRARY_SPECIFICATION is specified, \$GETQUI returns, as an RMS file name component, the name of the device control library for the specified queue. The library specification assumes the device and directory name SYS\$LIBRARY: and a file type of TLB. Since a file name can include up to 39 characters, the buffer length field of the item descriptor should specify 39 (bytes). This item code is meaningful only for output execution queues.

(QUI\$_DISPLAY_QUEUE item code)

QUI\$_LOG_QUEUE

When QUI\$_LOG_QUEUE is specified, \$GETQUI returns, as a character string, the name of the queue into which the log file produced for the specified batch job is to be entered for printing. This item code is applicable only to batch jobs. Since a queue name can contain up to 31 characters, the buffer length field of the item descriptor should specify 31 (bytes).

(QUI\$_DISPLAY_JOB item code)

QUI\$_LOG_SPECIFICATION

When QUI\$_LOG_SPECIFICATION is specified, \$GETQUI returns, as an RMS file specification, the name of the log file to be produced for the specified job. Since a file specification can include up to 255 characters, the buffer length field of the item descriptor should specify 255 (bytes). This item code is meaningful only for batch jobs.

The string returned is simply the log specification that was provided to the \$SNDJBC service to create the job. Therefore, to determine whether or not a log file is to be produced it is not sufficient to test this item code for a 0-length string; you need to examine the QUI\$V_JOB_LOG_NULL bit of the QUI\$_JOB_FLAGS item code.

(QUI\$_DISPLAY_JOB item code)

QUI\$_NOTE

When QUI\$_NOTE is specified, \$GETQUI returns, as a character string, the note that is to be printed on the job flag and file flag pages of the specified job. Since the note can include up to 255 characters, the buffer length field of the item descriptor should specify 255 (bytes). This item code is meaningful only for output execution queues.

(QUI\$_DISPLAY_JOB item code)

System Service Descriptions

\$GETQUI

QUI\$_OPERATOR_REQUEST

When QUI\$_OPERATOR_REQUEST is specified, \$GETQUI returns, as a character string, the message that is to be sent to the queue operator before the specified job begins to execute. Since the message can include up to 255 characters, the buffer length field of the item descriptor should specify 255 (bytes). This item code is meaningful only for output execution queues.

(QUI\$_DISPLAY_JOB item code)

QUI\$_OWNER_UIC

When QUI\$_OWNER_UIC is specified, \$GETQUI returns the owner UIC of the specified queue in standard longword format.

(QUI\$_DISPLAY_QUEUE item code)

QUI\$_PAGE_SETUP_MODULES

When QUI\$_PAGE_SETUP_MODULES is specified, \$GETQUI returns, as a comma-separated list, the names of the text modules that are to be extracted from the device control library and copied to the printer before each page of the specified form is printed. Since a text module name can include up to 31 characters and is separated from the previous text module name by a comma, the buffer length field of the item descriptor should specify 32 (bytes) for each possible text module. This item code is meaningful only for output execution queues.

(QUI\$_DISPLAY_FORM item code)

QUI\$_PARAMETER_1 through QUI\$_PARAMETER_8

When QUI\$_PARAMETER_1 through QUI\$_PARAMETER_8 are specified, \$GETQUI returns, as a character string, the value of the user-defined parameters that in batch jobs become the value of the DCL symbols P1 through P8 respectively. Since these parameters may include up to 255 characters the buffer length field of the item descriptor should specify 255 (bytes).

(QUI\$_DISPLAY_JOB item code)

QUI\$_PRIORITY

When QUI\$_PRIORITY is specified, \$GETQUI returns the scheduling priority of the specified job, which is a longword decimal value in the range 0 through 255.

Scheduling priority affects the order in which jobs assigned to a queue are initiated; it has no effect on the base execution priority of a job. The lowest scheduling priority value is 0, the highest is 255; that is, if a queue contains a job with a scheduling priority of 10 and a job with a scheduling priority of 2, the queue manager initiates the job with the scheduling priority of 10 first.

(QUI\$_DISPLAY_JOB item code)

QUI\$_PROCESSOR

When QUI\$_PROCESSOR is specified, \$GETQUI returns, as an RMS file name component, the name of the symbiont image that executes print jobs initiated from the specified queue. The file name assumes the device and directory name SYS\$SYSTEM; and the file type EXE. Since an RMS file name can include up to 39 characters the buffer length field of the item descriptor should specify 39 (bytes).

(QUI\$_DISPLAY_QUEUE item code)

System Service Descriptions

\$GETQUI

QUI\$_PROTECTION

When QUI\$_PROTECTION is specified, \$GETQUI returns, as a longword, the specified queue's protection mask. The protection mask is shown in the figure below.

Value change enable																Protection value															
WORLD				GROUP				OWNER				SYSTEM				WORLD				GROUP				OWNER				SYSTEM			
D	E	W	R	D	E	W	R	D	E	W	R	D	E	W	R	D	E	W	R	D	E	W	R	D	E	W	R	D	E	W	R
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

ZK-1724-84

Bits 0 through 15 specify the protection value: the four types of access (read, write, execute, delete) to be granted to the four classes of user (system, owner, group, world). Set bits deny access and clear bits allow access.

Bits 16 through 31 enable or disable bits 0 through 15. When a bit in the second word is set, the corresponding bit in the first word will affect the queue protection. When a bit in the second word is clear, the corresponding bit in the first word is ignored.

By default, the queue protection is (S:E,O:D,G:R,W:W).

(QUI\$_DISPLAY_QUEUE item code)

QUI\$_QUEUE_FLAGS

When QUI\$_QUEUE_FLAGS is specified, \$GETQUI returns, as a longword bit vector, the processing options that have been selected for the specified queue. Each processing option is represented by a bit. When \$GETQUI sets a bit, the jobs initiated from the queue are processed according to the corresponding processing option. Each bit in the vector has a symbolic name. These symbolic names are defined by the \$QUIDEF macro and are listed below.

Symbol	Description
QUI\$V_QUEUE_BATCH	Queue is a batch queue or a generic batch queue.
QUI\$V_QUEUE_CPU_DEFAULT	A default CPU time limit has been specified for all jobs in the queue.
QUI\$V_QUEUE_CPU_LIMIT	A maximum CPU time limit has been specified for all jobs in the queue.
QUI\$V_QUEUE_FILE_BURST	Burst and flag pages precede each file in each job initiated from the queue.
QUI\$V_QUEUE_FILE_BURST_ONE	Burst and flag pages precede only the first copy of the first file in each job initiated from the queue.
QUI\$V_QUEUE_FILE_FLAG	Flag page precedes each file in each job initiated from the queue.

System Service Descriptions

\$GETQUI

Symbol	Description
QUI\$V_QUEUE_FILE_FLAG_ONE	Flag page precedes only the first copy of the first file in each job initiated from the queue.
QUI\$V_QUEUE_FILE_PAGINATE	Output symbiont paginates output for each job initiated from this queue. The output symbiont paginates output by inserting a form feed whenever output reaches the bottom margin of the form.
QUI\$V_QUEUE_FILE_TRAILER	Trailer page follows each file in each job initiated from the queue.
QUI\$V_QUEUE_FILE_TRAILER_ONE	Trailer page follows only the last copy of the last file in each job initiated from the queue.
QUI\$V_QUEUE_GENERIC	The queue is a generic queue.
QUI\$V_QUEUE_GENERIC_SELECTION	The queue is an execution queue that can accept work from a generic queue.
QUI\$V_QUEUE_JOB_BURST	Burst and flag pages precede each job initiated from the queue.
QUI\$V_QUEUE_JOB_FLAG	A flag page precedes each job initiated from the queue.
QUI\$V_QUEUE_JOB_SIZE_SCHED	Jobs initiated from the queue are scheduled according to size, with the smallest job of a given priority processed first. Meaningful only for output queues.
QUI\$V_QUEUE_JOB_TRAILER	A trailer page follows each job initiated from the queue.
QUI\$V_QUEUE_RECORD_BLOCKING	The symbiont is permitted to concatenate, or block together, the output records it sends to the output device.
QUI\$V_QUEUE_RETAIN_ALL	All jobs initiated from the queue remain in the queue after they finish executing. Completed jobs are marked with a completion status.
QUI\$V_QUEUE_RETAIN_ERROR	Only jobs that do not complete successfully are retained in the queue.
QUI\$V_QUEUE_SWAP	Jobs initiated from the queue can be swapped.
QUI\$V_QUEUE_TERMINAL	The queue is a generic queue that can place jobs only in terminal queues.
QUI\$V_QUEUE_WSDEFAULT	Default working set size is specified for each job initiated from the queue.
QUI\$V_QUEUE_WSEXTENT	Working set extent is specified for each job initiated from the queue.
QUI\$V_QUEUE_WSQUOTA	Working set quota is specified for each job initiated from the queue.

(QUI\$_DISPLAY_QUEUE item code)

System Service Descriptions

\$GETQUI

QUI\$_QUEUE_NAME

When QUI\$_QUEUE_NAME is specified \$GETQUI returns, as a character string, the name of the specified queue or the name of the queue that contains the specified job. Since a queue name can include up to 31 characters, the buffer length field of the item descriptor should specify 31 (bytes).

(QUI\$_DISPLAY_JOB, QUI\$_DISPLAY_QUEUE item codes)

QUI\$_QUEUE_STATUS

When QUI\$_QUEUE_STATUS is specified, \$GETQUI returns the specified queue's status flags, which are contained in a longword bit vector. Listed below are the symbolic names for these flags, which are defined by the \$QUIDEF macro.

Symbol	Description
QUI\$_QUEUE_ALIGNING	Queue prints a specified amount of output so that paper can be properly aligned.
QUI\$_QUEUE_IDLE	Queue contains no job requests.
QUI\$_QUEUE_LOWERCASE	Queue is associated with a printer that can print uppercase and lowercase letters.
QUI\$_QUEUE_PAUSED	Execution of all current jobs in the queue is temporarily halted.
QUI\$_QUEUE_PAUSING	Queue is temporarily halting execution. Currently executing jobs are completing; temporarily, no new jobs can begin executing.
QUI\$_QUEUE_REMOTE	Queue is assigned to a physical device that is not connected to the local node.
QUI\$_QUEUE_RESETTING	Queue is resetting and stopping.
QUI\$_QUEUE_RESUMING	Queue is restarting after pausing.
QUI\$_QUEUE_SERVER	Queue processing is directed to a server symbiont.
QUI\$_QUEUE_STALLED	Physical device to which queue is assigned is stalled; that is, the device has not completed the last I/O request submitted to it.
QUI\$_QUEUE_STARTING	Queue is starting.
QUI\$_QUEUE_STOPPED	Queue is stopped.
QUI\$_QUEUE_STOPPING	Queue is stopping.
QUI\$_QUEUE_UNAVAILABLE	Physical device to which queue is assigned is not available.

(QUI\$_DISPLAY_QUEUE item code)

QUI\$_REQUEUE_QUEUE_NAME

When QUI\$_REQUEUE_QUEUE_NAME is specified, \$GETQUI returns, as a character string, the name of the queue to which the specified job is reassigned. Since a queue name can contain up to 31 characters, the buffer length field of the item descriptor should specify 31 (bytes).

(QUI\$_DISPLAY_JOB item code)

System Service Descriptions

\$GETQUI

QUI\$_SCSNODE_NAME

When QUI\$_SCSNODE_NAME is specified, \$GETQUI returns, as a character string, the 6-byte name of the VAX node on which jobs initiated from the specified queue execute. The node name matches the value of the SYSGEN parameter SCSNODE for the target node.

(QUI\$_DISPLAY_QUEUE item code)

QUI\$_SEARCH_FLAGS

QUI\$_SEARCH_FLAGS is an input value item code, which specifies a longword bit vector wherein each bit specifies the scope of \$GETQUI's search for objects specified in the call to \$GETQUI. The \$QUIDEF macro defines symbols for each option (bit) in the bit vector. The bit vector is constructed by specifying the symbolic names for the desired options in a logical OR operation. The following list contains the symbolic names for each option and the function code for which each flag is meaningful.

Symbol	Function Code	Description
QUI\$V_SEARCH_ALL_JOBS	QUI\$_DISPLAY_JOB	\$GETQUI searches all jobs included in the established queue context. If you do not specify this flag, \$GETQUI only returns information about jobs that have the same user name as the caller.
QUI\$V_SEARCH_BATCH	QUI\$_DISPLAY_QUEUE	Limits the scope of \$GETQUI's search to only batch queues.
QUI\$V_SEARCH_SYMBIONT	QUI\$_DISPLAY_QUEUE	Limits the scope of \$GETQUI's search to only output queues.
QUI\$V_SEARCH_THIS_JOB	QUI\$_DISPLAY_FILE QUI\$_DISPLAY_JOB QUI\$_DISPLAY_QUEUE	\$GETQUI returns information about the calling batch job, the command file being executed, or the queue associated with the calling batch job. \$GETQUI establishes a new queue and job context based on the job entry of the caller; this queue and job context is dissolved when \$GETQUI finishes executing. If you specify QUI\$V_SEARCH_THIS_JOB, \$GETQUI ignores all other QUI\$_SEARCH_FLAGS options.
QUI\$V_SEARCH_WILDCARD	QUI\$_DISPLAY_CHARACTERISTIC QUI\$_DISPLAY_FORM QUI\$_DISPLAY_QUEUE	\$GETQUI performs a search in wildcard mode even if QUI\$_SEARCH_NAME does not contain any wildcard characters.

QUI\$_SEARCH_NAME

QUI\$_SEARCH_NAME is an input value item code, which specifies, as a 1 to 31-character string, the name of the object about which \$GETQUI is to return information. The buffer must specify the name of a characteristic, form, or queue.

To direct \$GETQUI to perform a wildcard search, specify QUI\$_SEARCH_NAME as a string containing one or more of the wildcard characters % or *.

System Service Descriptions

\$GETQUI

QUI\$_SEARCH_NUMBER

QUI\$_SEARCH_NUMBER is an input value item code, which specifies, as a longword decimal value, the number of the form or job about which \$GETQUI is to return information. The buffer must specify a longword decimal value.

QUI\$_SUBMISSION_TIME

When QUI\$_SUBMISSION_TIME is specified, \$GETQUI returns, as a quadword absolute time value, the time at which the specified job was submitted to the queue.

(QUI\$_DISPLAY_JOB item code)

QUI\$_UIC

When QUI\$_UIC is specified, \$GETQUI returns, in standard longword format, the UIC of the owner of the specified job.

(QUI\$_DISPLAY_JOB item code)

QUI\$_USERNAME

When QUI\$_USERNAME is specified, \$GETQUI returns the username of the owner of the specified job. The username is returned as a 12-byte, zero-filled string.

(QUI\$_DISPLAY_JOB item code)

QUI\$_WSDEFAULT

When QUI\$_WSDEFAULT is specified, \$GETQUI returns the default working set size specified for the specified job or queue, which is a longword integer in the range 1 through 65,535. This value is meaningful only for batch jobs and execution and output queues.

(QUI\$_DISPLAY_JOB, QUI\$_DISPLAY_QUEUE item codes)

QUI\$_WSEXTENT

When QUI\$_WSEXTENT is specified, \$GETQUI returns the working set extent specified for the specified job or queue, which is a longword integer in the range 1 through 65,535. This value is meaningful only for batch jobs and execution and output queues.

(QUI\$_DISPLAY_JOB, QUI\$_DISPLAY_QUEUE item code)

QUI\$_WSQUOTA

When QUI\$_WSQUOTA is specified, \$GETQUI returns the working set quota for the specified job or queue, which is a longword integer in the range 1 through 65,535. This value is meaningful only for batch jobs and execution and output queues.

(QUI\$_DISPLAY_JOB, QUI\$_DISPLAY_QUEUE item code)

iosb

VMS Usage: **io_status_block**
type: **quadword (unsigned)**
access: **write only**
mechanism: **by reference**

I/O status block into which \$GETQUI writes the completion status after the requested operation has completed. The **iosb** is the address of the I/O status block.

System Service Descriptions

\$GETQUI

At request initiation \$GETQUI sets the value of the quadword I/O status block to 0. When the requested operation has completed, \$GETQUI writes a condition value in the first longword of the I/O status block. It writes the value 0 into the second longword; this longword is unused and reserved for future use.

The condition values returned by \$GETQUI in the I/O status block are condition values from the JBC facility, which are defined by the \$JBCMSGDEF macro. The condition values returned from the JBC facility are listed under the heading "Condition Values Returned in the I/O Status Block" which follows the Description section.

Though this argument is optional, DIGITAL strongly recommends that you specify it for the following reasons.

- If you are using an event flag to signal the completion of the service, you can test the I/O status block for a condition value to be sure that the event flag was not set by an event other than service completion.
- If you are using the \$SYNCH service to synchronize completion of the service, the I/O status block is a required argument for \$SYNCH.
- The condition value returned in R0 and the condition value returned in the I/O status block provide information about different aspects of the call to the \$GETQUI service. The condition value returned in R0 gives you information about the success or failure of the service call itself; the condition value returned in the I/O status block gives you information about the success or failure of the service operation. Therefore, to accurately assess the success or failure of the call to \$GETQUI, you must check the condition values returned in both R0 and the I/O status block.

astadr

VMS Usage: **ast_procedure**
type: **procedure entry mask**
access: **call without stack unwinding**
mechanism: **by reference**

AST service routine to be executed when \$GETQUI completes. The **astadr** argument is the address of the entry mask of this routine.

If specified, the AST routine executes at the same access mode as the caller of \$GETQUI.

astprm

VMS Usage: **user_parm**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

AST parameter to be passed to the AST service routine specified by the **astadr** argument. The **astprm** argument is this longword parameter.

DESCRIPTION

The caller must have READ access to the job or SYSPRV or OPER privilege to obtain job and file information. If the caller does not have privilege to access a job specified in a QUI\$_DISPLAY_JOB or QUI\$_DISPLAY_FILE operation, \$GETQUI returns a successful condition value. However, it sets the QUI\$_JOB_INACCESSIBLE bit of the QUI\$_JOB_STATUS item code and only returns information for the following item codes.

QUI\$_AFTER_TIME
QUI\$_COMPLETED_BLOCKS
QUI\$_ENTRY_NUMBER
QUI\$_INTERVENING_BLOCKS
QUI\$_INTERVENING_JOBS
QUI\$_JOB_SIZE
QUI\$_JOB_STATUS

\$GETQUI's Information Search

The \$GETQUI service returns information about objects contained in the system job queue file. Specifically, it returns information about the following types of object: characteristics, forms, queues, jobs contained in queues, and files contained in jobs that are located in queues.

When you call the \$GETQUI service, the job controller establishes an internal GETQUI context block (GQC). The system uses the GQC to temporarily store information and to keep track of its place in a wildcard sequence of operations. The system provides only one GQC per process, therefore only one \$GETQUI operation can be in progress at any one time.

To allow you to obtain information either about a particular object in a single call or about several objects in a sequence of calls, \$GETQUI supports three different search modes. The search modes, which are named below, affect the disposition of the GQC in different ways.

- **Nonwildcard Mode**—\$GETQUI returns information about a particular object in a single call. After the call completes, the system dissolves the GQC.
- **Wildcard Mode**—\$GETQUI returns information about several objects of the same type in a sequence of calls. The system saves the GQC between calls until the wildcard sequence completes.
- **Nested Wildcard Mode**—\$GETQUI returns information about the jobs contained in a selected queue or the files contained in a selected job in a sequence of calls. After each call, the system saves the GQC, so that the GQC can provide the queue or job context necessary for subsequent calls.

The sections that follow describe how each of the three search methods affects \$GETQUI's search for information; how you direct \$GETQUI to undertake each method; and how each method affects the contents of the GQC.

Nonwildcard Search Mode

In nonwildcard search mode, \$GETQUI can return information about the following objects.

- A specific characteristic or form definition that you identify by name or number.
- A specific queue definition that you specify by name.

System Service Descriptions

\$GETQUI

- A queue that is associated with a calling batch job, the calling batch job itself, or the command procedure file that the batch job is executing.

To obtain information about a specific characteristic or form definition, you need to specify the QUI\$_DISPLAY_CHARACTERISTIC (for a characteristic) or QUI\$_DISPLAY_FORM (for a form) function code. You also need to specify the name of the specific characteristic or form in the QUI\$_SEARCH_NAME item code or the number of the specific characteristic or form in the QUI\$_SEARCH_NUMBER item code. The name string you specify cannot include either of the wildcard characters * or %. You can specify both the QUI\$_SEARCH_NAME and QUI\$_SEARCH_NUMBER item codes, but the name and number you specify must be associated with the same characteristic or form definition.

To obtain information about a specific queue definition, specify the QUI\$_DISPLAY_QUEUE function code and provide the name of the queue in the QUI\$_SEARCH_NAME item code. The name-string you specify cannot include the wildcard characters * or %. You can restrict the scope of the search to a specific type of queue by selecting either the QUI\$_SEARCH_BATCH (to search batch queues) or QUI\$_SEARCH_SYMBIONT (to search output queues) option of the QUI\$_SEARCH_FLAGS item code. If the specified queue is not the type indicated in the QUI\$_SEARCH_FLAGS item code, \$GETQUI will be unable to locate the queue.

Finally, the \$GETQUI service provides an option that allows a batch job to obtain information about the queue, job, or command file that the associated batch job is executing without first entering nested wildcard mode to establish a queue or job context. You can make a call from the batch job that specifies the QUI\$_DISPLAY_QUEUE function code to obtain information about the queue from which the batch job was initiated; the QUI\$_DISPLAY_JOB function code to obtain information about the batch job itself; or the QUI\$_DISPLAY_FILE function code to obtain information about the files contained in the batch job. For each of these calls you must select the QUI\$_SEARCH_THIS_JOB option of the QUI\$_SEARCH_FLAGS item code. When you select this option, \$GETQUI ignores all other options in the QUI\$_SEARCH_FLAGS item code.

Wildcard Search Mode

In wildcard search mode, the system saves the GQC between calls to \$GETQUI, so that you can make a sequence of calls to \$GETQUI to get information about all of the characteristic, form, or queue definitions contained in the system job queue file.

To set up a wildcard search for characteristic or form definitions, specify the QUI\$_DISPLAY_CHARACTERISTIC (for a characteristic) or QUI\$_DISPLAY_FORM (for a form) function code; specify a name in the QUI\$_SEARCH_NAME item code that includes one or more wildcard characters (* or %).

To set up a wildcard search for queue definitions, specify the QUI\$_DISPLAY_QUEUE function code and specify a name in the QUI\$_SEARCH_NAME item code that includes one or more wildcard characters (* or %). You can indicate the type of the queue you want to search for by specifying the QUI\$_SEARCH_BATCH or QUI\$_SEARCH_SYMBIONT option of the QUI\$_SEARCH_FLAGS item code. If you select the QUI\$_SEARCH_BATCH option, \$GETQUI returns information only about batch queues; if you select the QUI\$_SEARCH_SYMBIONT option, \$GETQUI returns information only about output queues.

You can also force wildcard search mode for characteristic, form, or queue display operations by specifying the QUI\$V_SEARCH_WILDCARD option of the QUI\$_SEARCH_FLAGS item code. If you specify this option, the system saves the GQC between calls, even if you specify a name in the QUI\$_SEARCH_NAME item code that does not include any wildcard characters. This method of setting up a wildcard search is useful if you want to call \$GETQUI in a loop and exit when an unsuccessful condition value is returned. Whether or not you specify a wildcard name in the QUI\$_SEARCH_NAME item code, selecting the QUI\$V_SEARCH_WILDCARD option ensures that wildcard search mode is enabled.

Once you have set up a wildcard search, the wildcard search mode remains in operation until one of the following actions occurs, which causes the GQC to be released.

- \$GETQUI returns a JBC\$_NOMORExxx or JBC\$_NOSUCHxxx condition value on a call to display characteristic, form, or queue information, where "xxx" refers to CHAR, FORM, or QUE.
- You explicitly cancel the wildcard operation by specifying the QUI\$_CANCEL_OPERATION function code in a call to the \$GETQUI service.
- Your process terminates.

Nested Wildcard Search Mode

In nested wildcard search mode, the system saves the GQC between calls to \$GETQUI, so that you can make a sequence of calls to \$GETQUI to get information about jobs that are contained in a selected queue or files of the selected job. Two of \$GETQUI's operations, QUI\$_DISPLAY_JOB and QUI\$_DISPLAY_FILE, can only be used in a nested wildcard search, with one exception. The exceptional use of these two operations involves calls made to \$GETQUI from batch jobs; the exceptional use is described at the end of the "Nonwildcard Search Mode" section.

\$GETQUI needs to use the nested wildcard search mode because of the way in which \$GETQUI locates objects and the relationship among queues, jobs, and files. \$GETQUI performs only one operation per call. Thus, it can locate and return information about only one object in a single call. However, queues are objects that contain jobs and jobs are objects that contain files. Therefore, to get information about an object that is contained within another object, you must first make a call to \$GETQUI that specifies and locates the containing object and then make a call to request information about the contained object. The system saves the location of the containing object in the GQC.

For example, you must make at least two calls to \$GETQUI before you can call \$GETQUI to return file information. These calls establish the queue context and the job context for \$GETQUI's search for file information. In the first sequence of calls you specify the QUI\$_DISPLAY_QUEUE operation. You continue the first sequence until a call locates the queue that contains the job that contains the file of interest. This call establishes the queue context. In the second sequence of calls you specify the QUI\$_DISPLAY_JOB operation; these calls search the queue that was located in the first call sequence until a call locates the job that contains the file of interest. This call establishes the job context. Having located the queue and the job that contain the file, you can now use the QUI\$_DISPLAY_FILE operation to request file information.

System Service Descriptions

\$GETQUI

To set up a nested wildcard search for file or job information, perform a QUI\$_DISPLAY_QUEUE operation in wildcard search mode to establish the queue context necessary for the nested display job and file operations. You can enter the wildcard search mode for the display queue operation in two different ways: by specifying a wildcard name in the QUI\$_SEARCH_NAME item code; or by specifying a nonwildcard queue name and selecting the QUI\$_SEARCH_WILDCARD option of the QUI\$_SEARCH_FLAG item code. The second method of entering wildcard search mode is useful if you want to obtain information about one or more jobs or files within jobs for a specific queue and, therefore, want to specify a nonwildcard queue name, but still want to save the GQC once the queue context is established.

When you make a call to \$GETQUI that specifies the QUI\$_DISPLAY_JOB function code, by default \$GETQUI locates all the jobs in the selected queue that have the same username as the calling process. If you want to obtain information about all the jobs in the selected queue, select the QUI\$_SEARCH_ALL_JOBS option of the QUI\$_SEARCH_FLAGS item code.

Once you have established a queue context it remains in effect until you either change the context by making another call to \$GETQUI that specifies the QUI\$_DISPLAY_QUEUE function code, or one of the actions listed at the end of the "Wildcard Search Mode" section causes the GQC to be released. An established job context remains in effect until you change the context by making another call to \$GETQUI that specifies the QUI\$_DISPLAY_JOB function code or \$GETQUI returns a JBC\$_NOMOREJOB or JBC\$_NOSUCHJOB condition value. While the return of either of these two condition values releases the job context, the wildcard search remains in effect, because the GQC continues to maintain the queue context. Similarly, return of the JBC\$_NOMOREFILE or JBC\$_NOSUCHFILE condition value signals that there are no more files located in the current job context, however, these condition values do not affect the wildcard search mode, because the GQC continues to maintain the job context.

CONDITION VALUES RETURNED

SS\$_NORMAL	Successful completion.
SS\$_ACCVIO	The item list or input buffer cannot be read by the caller; or the return length buffer, output buffer, or status block cannot be written by the caller.
SS\$_BADPARAM	The function code is invalid; the item list contains an invalid item code; a buffer descriptor has an invalid length; or the reserved parameter has a nonzero value.
SS\$_DEVOFFLINE	The job controller process is not running.
SS\$_EXASTLM	The astadr argument was specified, and the process has exceeded its ASTLM quota.
SS\$_ILLEFC	The efn argument specifies an illegal event flag number.
SS\$_INSFMEM	The executive mode stack contains insufficient space to complete the request.
SS\$_MBFULL	The job controller mailbox is full.

System Service Descriptions

\$GETQUI

SS\$_MBTOOSML

The mailbox message is too large for the job controller mailbox.

SS\$_UNASEFC

The **efn** argument specifies an unassociated event flag cluster.

CONDITION VALUES RETURNED IN THE I/O STATUS BLOCK

JBC\$_NORMAL

Normal successful completion.

JBC\$_INVFUNCOD

The specified function code is invalid.

JBC\$_INVITMCO

The item list contains an invalid item code.

JBC\$_INVPARLEN

The length of a specified string is outside the valid range for that item code.

JBC\$_INVQUENAM

The queue name is not syntactically valid.

JBC\$_JOBQUEDIS

The request cannot be executed because the system job queue manager has not been started.

JBC\$_MISREQPAR

An item code that is required for the specified function code has not been specified.

JBC\$_NOJOBCTX

No job context has been established for a QUI\$_DISPLAY_FILE operation.

JBC\$_NOMORECHAR

There are no more characteristics defined; indicates the termination of a QUI\$_DISPLAY_CHARACTERISTIC wildcard operation.

JBC\$_NOMOREFILE

There are no more files associated with the current job context; indicates the termination of a QUI\$_DISPLAY_FILE wildcard operation for the current job context.

JBC\$_NOMOREFORM

There are no more forms defined; indicates the termination of a QUI\$_DISPLAY_FORM wildcard operation.

JBC\$_NOMOREJOB

There are no more jobs associated with the current job context; indicates the termination of a QUI\$_DISPLAY_JOB wildcard operation for the current queue context.

JBC\$_NOMOREQUE

There are no more queues defined; indicates the termination of a QUI\$_DISPLAY_QUEUE wildcard operation.

JBC\$_NOQUECTX

No queue context has been established for a QUI\$_DISPLAY_JOB or QUI\$_DISPLAY_FILE operation.

JBC\$_NOSUCHCHAR

The specified characteristic does not exist.

JBC\$_NOSUCHFILE

The specified file does not exist.

JBC\$_NOSUCHFORM

The specified form does not exist.

JBC\$_NOSUCHJOB

The specified job does not exist.

JBC\$_NOSUCHQUE

The specified queue does not exist.

System Service Descriptions

\$GETQUI

EXAMPLES

1

```
! Declare system service related symbols
INTEGER*4      SYS$GETQUIW,
2              STATUS
INCLUDE        '($QUIDEF)'

! Declare symbols for specific status code checking; currently JBC$
! completion codes cannot be defined locally with an INCLUDE statement
INTEGER*4      LIB$MATCH_COND
EXTERNAL       JBC$_NOSUCHJOB

! Define item list structure
STRUCTURE      /ITMLST/
  UNION
    MAP
      INTEGER*2 BUFLN, ITMCD
      INTEGER*4 BUFADR, RETADR
    END MAP
    MAP
      INTEGER*4 END_LIST
    END MAP
  END UNION
END STRUCTURE

! Define I/O status block structure
STRUCTURE      /IOSBLK/
  INTEGER*4     STS, ZEROED
END STRUCTURE

! Declare $GETQUIW item list and I/O status block
RECORD /ITMLST/ GETQUI_LIST(4)
RECORD /IOSBLK/ IOSB

! Declare variables used in $GETQUIW item list
CHARACTER*31   QUEUE_NAME
INTEGER*2      QUEUE_NAME_LEN
INTEGER*4      SEARCH_FLAGS,
2              ENTRY_NUMBER

! Initialize item list
GETQUI_LIST(1).BUFLN = 4
GETQUI_LIST(1).ITMCD = QUI$_SEARCH_FLAGS
GETQUI_LIST(1).BUFADR = %LOC(SEARCH_FLAGS)
GETQUI_LIST(1).RETADR = 0
GETQUI_LIST(2).BUFLN = 4
GETQUI_LIST(2).ITMCD = QUI$_ENTRY_NUMBER
GETQUI_LIST(2).BUFADR = %LOC(ENTRY_NUMBER)
GETQUI_LIST(2).RETADR = 0
GETQUI_LIST(3).BUFLN = 31
GETQUI_LIST(3).ITMCD = QUI$_QUEUE_NAME
GETQUI_LIST(3).BUFADR = %LOC(QUEUE_NAME)
GETQUI_LIST(3).RETADR = %LOC(QUEUE_NAME_LEN)
GETQUI_LIST(4).END_LIST = 0

SEARCH_FLAGS = QUI$_M_SEARCH_THIS_JOB

! Call $GETQUIW service to obtain job information
STATUS = SYS$GETQUIW (,
2                  %VAL(QUI$_DISPLAY_JOB),,
2                  GETQUI_LIST,
2                  IOSB,,)
IF (LIB$MATCH_COND (IOSB.STS, %LOC(JBC$_NOSUCHJOB))) THEN
  ! The search_this_job option can be used only by
  ! a batch job to obtain information about itself
  TYPE *, '<<< this job is not being run in batch mode >>>'
```

```

ENDIF
IF (STATUS) STATUS = IOSB.STS
IF (STATUS) THEN
    ! Display information
    TYPE *, 'Job entry number = ', ENTRY_NUMBER
    TYPE *, 'Queue name = ', QUEUE_NAME(1:QUEUE_NAME_LEN)
ELSE
    ! Signal error condition
    CALL LIB$SIGNAL (%VAL(STATUS))
ENDIF
END

```

.TITLE DEFINE_JBC_SYMBOLS

; Define JBC\$ completion codes here because they are not currently defined

; in the system object and shareable image libraries, nor in FORSYSDEF.TLB

\$JBCMSGDEF GLOBAL ; Keyword must be in uppercase

.END

The preceding FORTRAN program and accompanying MACRO routine demonstrates how a batch job can obtain information about itself from the system job queue file by using the \$GETQUIW system service. Use of the QUI\$M_SEARCH_THIS_OPTION in the QUI\$_SEARCH_FLAGS input item requires that the calling program run as a batch job, otherwise the \$GETQUIW service will return a JBC\$_NOSUCHJOB error.

2

```

! Declare system service related symbols
INTEGER*4    SYS$GETQUIW,
2            STATUS_Q,
2            STATUS_J,
2            NOACCESS
INCLUDE      '($QUIDEF)'

! Define item list structure
STRUCTURE    /ITMLST/
UNION
    MAP
        INTEGER*2 BUFLEN, ITMCOB
        INTEGER*4 BUFADR, RETADR
    END MAP
    MAP
        INTEGER*4 END_LIST
    END MAP
END UNION
END STRUCTURE

! Define I/O status block structure
STRUCTURE    /IOSBLK/
INTEGER*4    STS, ZEROED
END STRUCTURE

! Declare $GETQUIW item lists and I/O status block
RECORD /ITMLST/ QUEUE_LIST(4)
RECORD /ITMLST/ JOB_LIST(6)
RECORD /IOSBLK/ IOSB

! Declare variables used in $GETQUIW item lists
CHARACTER*31 SEARCH_NAME
CHARACTER*31 QUEUE_NAME
CHARACTER*30 JOB_NAME
CHARACTER*12 USERNAME
INTEGER*2    SEARCH_NAME_LEN,
2            QUEUE_NAME_LEN,
2            JOB_NAME_LEN,
2            USERNAME_LEN
INTEGER*4    SEARCH_FLAGS,
2            JOB_SIZE,
2            JOB_STATUS

! Solicit queue name to search; it may be a wildcard name
TYPE 9000
ACCEPT 9010, SEARCH_NAME_LEN, SEARCH_NAME

```

System Service Descriptions

\$GETQUI

```
! Initialize item list for the display queue operation
QUEUE_LIST(1).BUFLN = SEARCH_NAME_LEN
QUEUE_LIST(1).ITMCD = QUI$_SEARCH_NAME
QUEUE_LIST(1).BUFADR = %LOC(SEARCH_NAME)
QUEUE_LIST(1).RETADR = 0
QUEUE_LIST(2).BUFLN = 4
QUEUE_LIST(2).ITMCD = QUI$_SEARCH_FLAGS
QUEUE_LIST(2).BUFADR = %LOC(SEARCH_FLAGS)
QUEUE_LIST(2).RETADR = 0
QUEUE_LIST(3).BUFLN = 31
QUEUE_LIST(3).ITMCD = QUI$_QUEUE_NAME
QUEUE_LIST(3).BUFADR = %LOC(QUEUE_NAME)
QUEUE_LIST(3).RETADR = %LOC(QUEUE_NAME_LEN)
QUEUE_LIST(4).END_LIST = 0

! Initialize item list for the display job operation
JOB_LIST(1).BUFLN = 4
JOB_LIST(1).ITMCD = QUI$_SEARCH_FLAGS
JOB_LIST(1).BUFADR = %LOC(SEARCH_FLAGS)
JOB_LIST(1).RETADR = 0
JOB_LIST(2).BUFLN = 4
JOB_LIST(2).ITMCD = QUI$_JOB_SIZE
JOB_LIST(2).BUFADR = %LOC(JOB_SIZE)
JOB_LIST(2).RETADR = 0
JOB_LIST(3).BUFLN = 39
JOB_LIST(3).ITMCD = QUI$_JOB_NAME
JOB_LIST(3).BUFADR = %LOC(JOB_NAME)
JOB_LIST(3).RETADR = %LOC(JOB_NAME_LEN)
JOB_LIST(4).BUFLN = 12
JOB_LIST(4).ITMCD = QUI$_USERNAME
JOB_LIST(4).BUFADR = %LOC(USERNAME)
JOB_LIST(4).RETADR = %LOC(USERNAME_LEN)
JOB_LIST(5).BUFLN = 4
JOB_LIST(5).ITMCD = QUI$_JOB_STATUS
JOB_LIST(5).BUFADR = %LOC(JOB_STATUS)
JOB_LIST(5).RETADR = 0
JOB_LIST(6).END_LIST = 0

! Request search of all jobs present in output queues; also force
! wildcard mode to maintain the internal search context block after
! the first call when a non-wild queue name is entered--this preserves
! queue context for the subsequent display job operation
SEARCH_FLAGS = (QUI$_SEARCH_WILDCARD .OR.
2             QUI$_SEARCH_SYMBIONT .OR.
2             QUI$_SEARCH_ALL_JOBS)

! Dissolve any internal search context block for the process
STATUS_Q = SYS$GETQUIW (,%VAL(QUI$_CANCEL_OPERATION),....)

! Locate next output queue; loop until an error status is returned
DO WHILE (STATUS_Q)
    STATUS_Q = SYS$GETQUIW (,
2        %VAL(QUI$_DISPLAY_QUEUE),,
2        QUEUE_LIST,
2        IOSB,,)
    IF (STATUS_Q) STATUS_Q = IOSB.STS
    IF (STATUS_Q) TYPE 9020, QUEUE_NAME(1:QUEUE_NAME_LEN)
    STATUS_J = 1
```

System Service Descriptions

\$GETQUI

```
! Get information on next job in queue; loop until error return
DO WHILE (STATUS_Q .AND. STATUS_J)
  STATUS_J = SYS$GETQUIW (,
2      %VAL(QUI$_DISPLAY_JOB),,
2      JOB_LIST,
2      IOSB,,)
  IF (STATUS_J) STATUS_J = IOSB.STS
  IF ((STATUS_J) .AND. (JOB_SIZE .GE. 500)) THEN
    NOACCESS = (JOB_STATUS .AND. QUI$_JOB_INACCESSIBLE)
    IF (NOACCESS .NE. 0) THEN
      TYPE 9030, JOB_SIZE
    ELSE
      TYPE 9040, JOB_SIZE,
2      USERNAME(1:USERNAME_LEN),
2      JOB_NAME(1:JOB_NAME_LEN)
    ENDIF
  ENDIF
ENDDO
ENDDO
9000 FORMAT (' Enter queue name to search: ', $)
9010 FORMAT (Q, A31)
9020 FORMAT ('Queue name = ', A)
9030 FORMAT (' Job size = ', I5, ' <no read access privilege>')
9040 FORMAT (' Job size = ', I5,
2      ' Username = ', A, T46,
2      ' Job name = ', A)
END
```

The preceding FORTRAN program demonstrates how any job can obtain information about other jobs from the system job queue file by using the \$GETQUIW system service. This program lists all print jobs in output queues with a job size of 500 blocks or more. It also displays queue name, job size, user name, and job name information for each job listed.



\$GETQUIW—Get Queue Information and Wait for Completion

The Get Queue Information and Wait for Completion service (\$GETQUIW) returns information about queues and jobs initiated from those queues. The \$SNDJBC service is the major interface to the VAX/VMS Job Controller, which is the VAX/VMS queue and accounting manager. See the "Description" section of the \$SNDJBC service for a discussion of the different types of job and queue.

The \$GETQUIW service completes synchronously; that is, it returns to the caller with the requested information. For asynchronous completion use the Get Queue Information (\$GETQUI) service; \$GETQUI returns to the caller after queuing the information request, without waiting for the information to be returned.

In all other respects, \$GETQUIW is identical to \$GETQUI. Refer to the documentation of \$GETQUI for all other information about the \$GETQUIW service. For additional information about system service completion, refer to the Synchronize (\$SYNCH) service.

FORMAT

SYSS\$GETQUIW *[efn],func[,nullarg][,itmlst][,iosb]
[,astadr][,astprm]*

System Service Descriptions

\$GETSYI

\$GETSYI—Get System-Wide Information

The Get System-Wide Information service returns information about the local VAX system or about other VAX systems in a cluster.

For Version 4.0 of VAX/VMS, both the \$GETSYI service and the Get System-Wide Information and Wait (\$GETSYIW) service complete synchronously; that is, they return to the caller with the requested information.

However, in the future, \$GETSYI will be modified to complete asynchronously; that is, it will return to the caller after queuing the information request, without waiting for the information to be returned. For this reason, DIGITAL recommends that you use the \$GETSYIW service for synchronous completion.

For additional information about system service completion, refer to the Synchronize (\$SYNCH) service and to Section 2.5.

FORMAT	SYS\$GETSYI <i>[efn]</i> , <i>[csidadr]</i> , <i>[nodename]</i> , <i>itmlst</i> , <i>[iosb]</i> , <i>[astadr]</i> , <i>[astprm]</i>
---------------	---

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS *efn*

VMS Usage: **ef_number**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Number of the event flag to be set when the \$GETSYI request completes. The *efn* argument is a longword containing this number.

Upon request initiation, \$GETSYI clears the specified event flag (or event flag 0 if *efn* was not specified). Then when the request completes, the specified event flag (or event flag 0) is set.

csidadr

VMS Usage: **process_id**
type: **longword (unsigned)**
access: **modify**
mechanism: **by reference**

Cluster system identification of the VAX node about which \$GETSYI is to return information. The *csidadr* is the address of a longword containing this identification value.

System Service Descriptions

\$GETSYI

The cluster system identification of a VAX node is assigned by the cluster-connection software and may be obtained by the DCL command SHOW CLUSTER. The value of the cluster system identification for a VAX node is not permanent; a new value is assigned to a VAX node whenever it joins or rejoins the VAXcluster.

A VAX node may also be specified to \$GETSYI by using the **nodename** argument. If **csidadr** is specified, then **nodename** need not be, and vice versa. If both are specified, they must identify the same VAX node. If neither is specified, \$GETSYI returns information about the local VAX node. However, for wildcard operations, the **csidadr** argument must be used.

If **csidadr** is specified as -1, \$GETSYI assumes a wildcard operation and returns the requested information for each VAX node in the cluster, one node per call. In this case, the program should test for the condition value **SS\$_NOMORENODE** after each call to \$GETSYI and should stop calling \$GETSYI when **SS\$_NOMORENODE** is returned.

nodename

VMS Usage: **process_name**

type: **character-coded text string**

access: **read only**

mechanism: **by descriptor—fixed length string descriptor**

Name of the VAX node about which \$GETSYI is to return information. The **nodename** argument is the address of a character string descriptor pointing to this name string.

The node name string must contain from 1 to 15 characters and must correspond exactly to the VAX node name; no trailing blanks or abbreviations are permitted.

A VAX node may also be specified to \$GETSYI by using the **csidadr** argument. See the description of **csidadr**.

itmlst

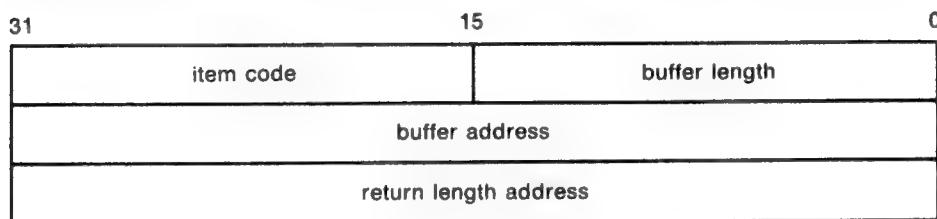
VMS Usage: **item_list_3**

type: **longword (unsigned)**

access: **read only**

mechanism: **by reference**

Item list specifying which information about the VAX node(s) is to be returned. The **itmlst** argument is the address of a list of item descriptors, each of which describes an item of information. The list of item descriptors is terminated by a longword of 0. The following diagram depicts a single item descriptor.



ZK-1705-84

System Service Descriptions

\$GETSYI

\$GETSYI Item Descriptor Fields

buffer length

A word containing a user-supplied integer specifying the length (in bytes) of the buffer in which \$GETSYI is to write the information. The length of the buffer needed depends upon the item code specified in the **item code** field of the item descriptor. If the value of **buffer length** is too small, \$GETSYI truncates the data.

item code

A word containing a user-supplied symbolic code specifying the item of information that \$GETSYI is to return. These codes are defined by the \$SYIDEF macro. A description of each item code is given in the "\$GETSYI Item Codes" section.

buffer address

A longword containing the user-supplied address of the buffer in which \$GETSYI is to write the information.

return length address

A longword containing the user-supplied address of a word in which \$GETSYI writes the length in bytes of the information it actually returned.

\$GETSYI Item Codes

SYI\$_BOOTTIME

When SYI\$_BOOTTIME is specified, \$GETSYI returns the time when the VAX node was booted. \$GETSYI returns this information only for the local VAX node.

Since the returned time is in the standard 64-bit absolute time format, the **buffer length** field in the item descriptor should specify 8 (bytes).

SYI\$_CHARACTER_EMULATED

When SYI\$_CHARACTER_EMULATED is specified, \$GETSYI returns the number 1 if the character string instructions are emulated on the CPU and 0 if they are not. \$GETSYI returns this information only for the local VAX node.

Since this number is a Boolean value (1 or 0), the **buffer length** field in the item descriptor should specify 1 (byte).

SYI\$_CLUSTER_FSYSID

When SYI\$_CLUSTER_FSYSID is specified, \$GETSYI returns the system identification of the founding node, which is the first node in the cluster to boot.

This system identification is assigned to the node by the cluster management software and may be obtained by using the DCL command SHOW CLUSTER. Since the system identification is a 6-byte hexadecimal number, the **buffer length** field in the item descriptor should specify 6 (bytes).

SYI\$_CLUSTER_FTIME

When SYI\$_CLUSTER_FTIME is specified, \$GETSYI returns the time when the founding node booted. The founding node is the first node in the cluster to boot.

Since the returned time is in the standard 64-bit absolute time format, the **buffer length** field in the item descriptor should specify 8 (bytes).

SYI\$_CLUSTER_MEMBER

When SYI\$_CLUSTER_MEMBER is specified, \$GETSYI returns the membership status of the node in the cluster. The membership status specifies whether the node is or is not currently a member of the cluster.

Since the membership status of a node is described in a 1-byte bit field, the **buffer length** field in the item descriptor should specify 1 (byte). If bit 0 in the bit field is set, the node is a member of the cluster; if it is clear, then it is not a member of the cluster.

SYI\$_CLUSTER_NODES

When SYI\$_CLUSTER_NODES is specified, \$GETSYI returns the number (in decimal) of nodes currently in the cluster.

Since this number is a word in length, the **buffer length** field in the item descriptor should specify 2 (bytes).

SYI\$_CLUSTER_QUORUM

When SYI\$_CLUSTER_QUORUM is specified, \$GETSYI returns the number (in decimal) that is the total of the quorum values held by all nodes in the cluster. Each node's quorum value is determined by its SYSGEN parameter QUORUM.

Since this number is a word in length, the **buffer length** field in the item descriptor should specify 2 (bytes).

SYI\$_CLUSTER_VOTES

When SYI\$_CLUSTER_VOTES is specified, \$GETSYI returns the total number of votes held by all nodes in the cluster. The number of votes held by any one node is determined by that node's SYSGEN parameter VOTES.

Since this decimal number is a word in length, the **buffer length** field in the item descriptor should specify 2 (bytes).

SYI\$_CPU

When SYI\$_CPU is specified, \$GETSYI returns the CPU processor type of the node. \$GETSYI returns this information only for the local VAX node.

Since the processor type is a longword decimal number, the **buffer length** field in the item descriptor should specify 4 (bytes).

Symbols for the processor types are defined by the \$PRDEF macro. The following gives the current processors and their symbols.

Processor	Symbol
VAX-11 780, 782, 785	PR\$_SID_TYP780
VAX-11 750	PR\$_SID_TYP750
VAX-11 730	PR\$_SID_TYP730
MICROVAX	PR\$_SID_TYPUV1

SYI\$_DECIMAL_EMULATED

When SYI\$_DECIMAL_EMULATED is specified, \$GETSYI returns the number 1 if the decimal string instructions are emulated on the CPU and a 0 if they are not. \$GETSYI returns this information only for the local VAX node.

Since this number is a Boolean value (1 or 0), the **buffer length** field in the item descriptor should specify 1 (byte).

System Service Descriptions

\$GETSYI

SYI\$_D_FLOAT_EMULATED

When SYI\$_D_FLOAT_EMULATED is specified, \$GETSYI returns the number 1 if the D_floating instructions are emulated on the CPU and a 0 if they are not. \$GETSYI returns this information only for the local VAX node.

Since this number is a Boolean value (1 or 0), the **buffer length** field in the item descriptor should specify 1 (byte).

SYI\$_F_FLOAT_EMULATED

When SYI\$_F_FLOAT_EMULATED is specified, \$GETSYI returns the number 1 if the F_floating instructions are emulated on the CPU and a 0 if they are not. \$GETSYI returns this information only for the local VAX node.

Since this number is a Boolean value (1 or 0), the **buffer length** field in the item descriptor should specify 1 (byte).

SYI\$_G_FLOAT_EMULATED

When SYI\$_G_FLOAT_EMULATED is specified, \$GETSYI returns the number 1 if the G_floating instructions are emulated on the CPU and a 0 if they are not. \$GETSYI returns this information only for the local VAX node.

Since this number is a Boolean value (1 or 0), the **buffer length** field in the item descriptor should specify 1 (byte).

SYI\$_H_FLOAT_EMULATED

When SYI\$_H_FLOAT_EMULATED is specified, \$GETSYI returns the number 1 if the H_floating instructions are emulated on the CPU and a 0 if they are not. \$GETSYI returns this information only for the local VAX node.

Since this number is a Boolean value (1 or 0), the **buffer length** field in the item descriptor should specify 1 (byte).

SYI\$_NODE_AREA

When SYI\$_NODE_AREA is specified, \$GETSYI returns the DECNET area of the node.

Since the DECNET area is a longword decimal number, the **buffer length** field in the item descriptor should specify 4 (bytes).

SYI\$_NODE_CSID

When SYI\$_NODE_CSID is specified, \$GETSYI returns the cluster system id (CSID) of the VAX node. The CSID is a longword hexadecimal number assigned to the node by the cluster management software.

Since the CSID is a longword, the **buffer length** field in the item descriptor should specify 4 (bytes).

SYI\$_NODE_HWTYPE

When SYI\$_NODE_HWTYPE is specified, \$GETSYI returns the hardware type of the node. The hardware type of a node indicates whether the node is a VAX-11 785, VAX-11 780, VAX-11 750, or HSC storage controller.

Since the hardware type is a 4-byte ASCII string, the **buffer length** field in the item descriptor should specify 4 (bytes).

SYI\$_NODE_HWVERS

When SYI\$_NODE_HWVERS is specified, \$GETSYI returns the hardware version of the node.

Since the hardware version is a 12-byte hexadecimal number, the **buffer length** field in the item descriptor should specify 12 (bytes).

System Service Descriptions

\$GETSYI

SYIS_NODE_NUMBER

When SYIS_NODE_NUMBER is specified, \$GETSYI returns the DECNET number of the node.

Since the DECNET number is a longword decimal number, the **buffer length** field in the item descriptor should specify 4 (bytes).

SYIS_NODE_QUORUM

When SYIS_NODE_QUORUM is specified, \$GETSYI returns the value (in decimal) of the quorum held by the node. This number is determined by the node's SYSGEN parameter QUORUM.

Since this number is a word in length, the **buffer length** field in the item descriptor should specify 2 (bytes).

SYIS_NODE_SWINCARN

When SYIS_NODE_SWINCARN is specified, \$GETSYI returns the software incarnation of the node.

Since the software incarnation of the node is an 8-byte hexadecimal number, the **buffer length** field in the item descriptor should specify 8 (bytes).

SYIS_NODE_SWTYPE

When SYIS_NODE_SWTYPE is specified, \$GETSYI returns the software type of the node. The software type indicates whether the node is a VAX/VMS system or an HSC storage controller.

Since the software type is a 4-byte ASCII string, the **buffer length** field in the item descriptor should specify 4 (bytes).

SYIS_NODE_SWVERS

When SYIS_NODE_SWVERS is specified, \$GETSYI returns the software version of the node.

Since the software version is a 4-byte ASCII string, the **buffer length** field in the item descriptor should specify 4 (bytes).

SYIS_NODE_SYSTEMID

When SYIS_NODE_SYSTEMID is specified, \$GETSYI returns the system identification of the node.

This system identification is assigned to the node by the cluster management software and may be obtained by using the DCL command SHOW CLUSTER. Since the system identification is a 6-byte hexadecimal number, the **buffer length** field in the item descriptor should specify 6 (bytes).

SYIS_NODE_VOTES

When SYIS_NODE_VOTES is specified, \$GETSYI returns the number (in decimal) of votes held by the node. This number is determined by the node's SYSGEN parameter VOTES.

Since this number is a word in length, the **buffer length** field in the item descriptor should specify 2 (bytes).

SYIS_NODENAME

When SYIS_NODENAME is specified, \$GETSYI returns, as a character string, the name of the node in the returned length area specified in the item list.

Since this name can include up to 15 characters, the **buffer length** field in the item descriptor should specify 15 (bytes).

System Service Descriptions

\$GETSYI

SYI\$_PAGEFILE_FREE

When SYI\$_PAGEFILE_FREE is specified, \$GETSYI returns the number of free pages in the currently installed paging files. \$GETSYI returns this information only for the local VAX node.

Since this number is a longword, the **buffer length** field in the item descriptor should specify 4 (bytes).

SYI\$_PAGEFILE_PAGE

When SYI\$_PAGEFILE_PAGE is specified, \$GETSYI returns the number of pages in the currently installed paging files. \$GETSYI returns this information only for the local VAX node.

Since this number is a longword, the **buffer length** field in the item descriptor should specify 4 (bytes).

SYI\$_SCS_EXISTS

When SYI\$_SCS_EXISTS is specified, \$GETSYI returns a longword value that is interpreted as Boolean. If the value is 1, the System Communication Subsystem (SCS) is currently loaded on the VAX node; if the value is 0, the SCS is not currently loaded.

SYI\$_SID

When SYI\$_SID is specified, \$GETSYI returns the contents of the system identification register of the VAX node. For more information on the meaning of the contents of the system identification register, see the *VAX Hardware Handbook*. \$GETSYI returns this information only for the local VAX node.

Since the value of this register is a longword hexadecimal number, the **buffer length** field in the item descriptor should specify 4 (bytes).

SYI\$_SWAPFILE_FREE

When SYI\$_SWAPFILE_FREE is specified, \$GETSYI returns the number of free pages in the currently installed swapping files. \$GETSYI returns this information only for the local VAX node.

Since this number is a longword, the **buffer length** field in the item descriptor should specify 4 (bytes).

SYI\$_SWAPFILE_PAGE

When SYI\$_SWAPFILE_PAGE is specified, \$GETSYI returns the number of pages in the currently installed swapping files. \$GETSYI returns this information only for the local VAX node.

Since this number is a longword, the **buffer length** field in the item descriptor should specify 4 (bytes).

SYI\$_VERSION

When SYI\$_VERSION is specified, \$GETSYI returns, as a character string, the software version number of the VAX/VMS operating system that is running on the VAX node. \$GETSYI returns this information only for the local VAX node.

Since the version number is an 8-byte blank-filled string, the **buffer length** field in the item descriptor should specify 8 (bytes).

SYI\$_xxxx

When SYI\$_xxxx is specified, \$GETSYI returns the current value of the SYSGEN parameter named "xxxx" for the VAX node. \$GETSYI returns this information only for the local VAX node.

The buffer must specify a longword into which \$GETSYI will write the value of the specified SYSGEN parameter. For a list and description of all system parameters, refer to Table SGN-2 in the *VAX/VMS Utilities Reference Volume*.

iosb

VMS Usage: **io_status_block**
type: **quadword (unsigned)**
access: **write only**
mechanism: **by reference**

I/O status block which is to receive the final completion status. The **iosb** is the address of the quadword I/O status block.

When the **iosb** argument is specified, \$GETSYI sets the quadword to zero upon request initiation. Upon request completion, a condition value is returned to the first longword; the second longword is reserved to DIGITAL.

Though this argument is optional, DIGITAL strongly recommends that you specify it for the following reasons.

- If you are using an event flag to signal the completion of the service, you can test the I/O status block for a condition value to be sure that the event flag was not set by an event other than service completion.
- If you are using the \$SYNCH service to synchronize completion of the service, the I/O status block is a required argument for \$SYNCH.
- The condition value returned in R0 and the condition value returned in the I/O status block provide information about different aspects of the call to the \$GETSYI service. The condition value returned in R0 gives you information about the success or failure of the service call itself; the condition value returned in the I/O status block gives you information about the success or failure of the service operation. Therefore, to accurately assess the success or failure of the call to \$GETSYI, you must check the condition values returned in both R0 and the I/O status block.

astadr

VMS Usage: **ast_procedure**
type: **procedure entry mask**
access: **call without stack unwinding**
mechanism: **by reference**

AST service routine to be executed when \$GETSYI completes. The **astadr** is the address of the entry mask of this routine.

If **astadr** is specified, the AST routine will execute at the same access mode as the caller of the \$GETSYI service.

astprm

VMS Usage: **user_arg**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

AST parameter to be passed to the AST service routine specified by the **astadr** argument. The **astprm** argument is the longword parameter.

System Service Descriptions

\$GETSYI

DESCRIPTION This service uses the process's AST limit quota (ASTLM).

CONDITION VALUES RETURNED

SS\$_NORMAL	Service successfully completed.
SS\$_NOMORENODE	A wildcard operation was requested, and \$GETSYI has returned information about all available VAX nodes.
SS\$_BADPARAM	The item list contains an invalid item code.
SS\$_ACCVIO	The caller cannot read the item list; cannot write to the buffer specified by the buffer address field in an item descriptor; or cannot write to the return length address field in an item descriptor.
SS\$_EXASTLM	The process has exceeded its AST limit quota.
SS\$_NOSUCHNODE	The specified VAX node does not exist or is not currently a member of the VAXcluster.

CONDITION VALUES RETURNED IN THE I/O STATUS BLOCK

Same as those returned in R0.

EXAMPLES

□

```
! Declare system service related symbols
INTEGER*4    SYS$GETSYIW,
2            STATUS
! External declaration is an alternative to including $SYIDF
EXTERNAL     SYI$_VERSION,
2            SYI$_NODENAME
! Define item list structure
STRUCTURE    /ITMLST/
UNION
  MAP
    INTEGER*2 BUFLN
    INTEGER*2 ITMCD
    INTEGER*4 BUFADR
    INTEGER*4 RETADR
  END MAP
  MAP
    INTEGER*4 END_LIST
  END MAP
END UNION
END STRUCTURE
! Define I/O status block structure
STRUCTURE    /IOSBLK/
INTEGER*4    STS, RESERVED
END STRUCTURE
```

System Service Descriptions

\$GETSYI

```
! Declare $GETSYIW item list and I/O status block
RECORD /ITMLST/ GETSYI_LIST(3)
RECORD /IOBBLK/ IOB

! Declare variables used in $GETSYIW item list
CHARACTER*8      VERSION
CHARACTER*16     NODENAME
INTEGER*2        VERSION_LEN,
2               NODENAME_LEN

! Initialize item list
GETSYI_LIST(1).BUFLEN = 8
GETSYI_LIST(1).ITMCD = %LOC(SYI%_VERSION)
GETSYI_LIST(1).BUFADR = %LOC(VERSION)
GETSYI_LIST(1).RETADR = %LOC(VERSION_LEN)
GETSYI_LIST(2).BUFLEN = 16
GETSYI_LIST(2).ITMCD = %LOC(SYI%_NODENAME)
GETSYI_LIST(2).BUFADR = %LOC(NODENAME)
GETSYI_LIST(2).RETADR = %LOC(NODENAME_LEN)
GETSYI_LIST(3).END_LIST = 0

! Display the system version number string
STATUS = SYS$GETSYIW (,,GETSYI_LIST,IOB,..)
IF (STATUS) STATUS = IOB.STS
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
TYPE *, 'System version is ', VERSION(1:VERSION_LEN)
END
```

The preceding FORTRAN program demonstrates the use of the \$GETSYIW service to obtain the operating system version number string and the system's node name.

System Service Descriptions

\$GETSYIW

\$GETSYIW—Get System-Wide Information and Wait

The Get System-Wide Information and Wait service returns information about the local VAX system or about other VAX systems in a cluster.

For Version 4.0 of VAX/VMS, the \$GETSYIW service is identical to the Get System-Wide Information (\$GETSYI) service. Both services return the same information, and both complete synchronously; that is, they return to the caller with the requested information.

However, in the future the \$GETSYI service will be modified to complete asynchronously; that is, it will return to the caller after queuing the information request, without waiting for the information to be returned. For this reason, DIGITAL recommends that you use the \$GETSYIW service for synchronous completion.

Refer to the documentation of \$GETSYI for all other information about the \$GETSYIW service.

For additional information about system service completion, refer to the Synchronize (\$SYNCH) service and to Section 2.5.

FORMAT

SYS\$GETSYIW *[efn] , [csidadr] , [nodename] , itmlst
[iosb] [,astadr] [,astprm]*

Either the **csidadr** or the **nodename** argument must be specified, but not both. However, the **csidadr** argument must be used for wildcard operations.

\$GETTIM—Get Time

The Get Time service returns the current system time in 64-bit format.

FORMAT **SYSS\$GETTIM** *timadr*

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENT *timadr*

VMS Usage: **date_time**
type: **quadword (unsigned)**
access: **write only**
mechanism: **by reference**

Address of a quadword that is to receive the current time in 64-bit format.

DESCRIPTION

The system time is updated every 10 milliseconds, and the time is returned in 100-nanosecond units from the system base time.

See Section 9 for additional information about the system time.

**CONDITION
VALUES
RETURNED**

SS\$_NORMAL
SS\$_ACCVIO

Service successfully completed.

The quadword to receive the time cannot be written by the caller.

\$GETUAI

The Get User Authorization Information (\$GETUI) service returns authorization information about a specified user.

SYSS\$GETUAI *[nullarg],[nullarg],usrnam,itmlst*
,[nullarg],[nullarg],[nullarg]

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

nullarg

VMS Usage: null_arg
type: longword (unsigned)
access: read only
mechanism: by value

Place-holding argument. This argument is reserved to DIGITAL.

usrnam

VMS Usage: **char_string**
type: **character-coded text string**
access: **read only**
mechanism: **by descriptor—fixed length string descriptor**

Name of the user about whom `$GETUAI` returns authorization information. The **usrnam** argument is the address of a descriptor pointing to a character text string containing the user name. The user name string may contain a maximum of 12 alphanumeric characters.

itmlst

VMS Usage: **item_list_3**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Item list specifying which information from the specified user's UAF (user authorization file) record is to be returned. The **itmlst** argument is the address of a list of one or more item descriptors, each of which specifies an item code. The item list is terminated by an item code of 0 or by a longword of 0. The following diagram depicts the structure of a single item descriptor.

System Service Descriptions

\$GETUAI

31	15	0
item code		buffer length
buffer address		
return length address		

ZK-1705-84

\$GETUAI Item Descriptor Fields

buffer length

A word specifying the length (in bytes) of the buffer in which \$GETUAI is to write the information. The length of the buffer varies depending on the item code specified in the **item code** field of the item descriptor and is given in the description of each item code. If the value of **buffer length** is too small, \$GETUAI truncates the data.

item code

A word containing a user-supplied symbolic code specifying the item of information that \$GETUAI is to return. These codes are defined by the \$UAIDEF macro and have the format: `UAI$_code`. Each item code is described below.

buffer address

A longword containing the user-supplied address of the buffer in which \$GETUAI is to write the information.

return length address

A longword containing the user-supplied address of a word in which \$GETUAI writes the length in bytes of the information it actually returned.

\$GETUAI Item Codes

UAI\$_ACCOUNT

When `UAI$_ACCOUNT` is specified, \$GETUAI returns, as a blank-filled character string, the account name of the user. Since an account name can include up to 8 characters plus a size-byte prefix, the buffer length field of the item descriptor should specify 9 (bytes).

UAI\$_ASTLM

When `UAI$_ASTLM` is specified, \$GETUAI returns the AST queue limit, which is a longword integer in the range 1 through 65,535.

UAI\$_BATCH_ACCESS_P

When `UAI$_BATCH_ACCESS_P` is specified, \$GETUAI returns the range of times during which batch access is permitted for primary days. Each bit set in the longword represents a one-hour period, from bit 0 as midnight to 1 AM to bit 23 as 11 PM to midnight.

UAI\$_BATCH_ACCESS_S

When `UAI$_BATCH_ACCESS_S` is specified, \$GETUAI returns the range of times during which batch access is permitted for secondary days. Each bit set in the longword represents a one-hour period, from bit 0 as midnight to 1 AM to bit 23 as 11 PM to midnight.

System Service Descriptions

\$GETUAI

UAI\$_BIOLM

When UAI\$_BIOLM is specified, \$GETUAI returns the buffered I/O count limit, which is a longword integer in the range 1 through 65,535.

UAI\$_BYTLM

When UAI\$_BYTLM is specified, \$GETUAI returns the buffered I/O byte limit, which is a longword integer in the range 1 through 65,535.

UAI\$_CLITABLES

When UAI\$_CLITABLES is specified, \$GETUAI returns, as a character string, the name of the user-defined CLI table for the account, if any. Since the CLI table name can include up to 31 characters plus a size-byte prefix, the buffer length field of the item descriptor should specify 32 (bytes).

UAI\$_CPUTIM

When UAI\$_CPUTIM is specified, \$GETUAI returns, as a longword decimal value, the maximum CPU time limit (per session) for the process in 10-millisecond units.

UAI\$_DEFCLI

When UAI\$_DEFCLI is specified, \$GETUAI returns, as an RMS file name component, the name of the command language interpreter used to execute the specified batch job. The file specification returned assumes the device name and directory SYS\$SYSTEM: and the file type EXE. Since a file name can include up to 39 characters plus a size-byte prefix, the buffer length field in the item descriptor should specify 40 (bytes).

UAI\$_DEFDEV

When UAI\$_DEFDEV is specified, \$GETUAI returns, as a 1- to 15-character string, the name of the default device. Since the device name string can include up to 15 characters plus a size-byte prefix, the buffer length field in the item descriptor should specify 16 (bytes).

UAI\$_DEFDIR

When UAI\$_DEFDIR is specified, \$GETUAI returns, as a 1- to 63-character string, the name of the default directory. Since the directory name string can include up to 63 characters plus a size-byte prefix, the buffer length field in the item descriptor should specify 64 (bytes).

UAI\$_DEF_PRIV

When UAI\$_DEF_PRIV is specified, \$GETUAI returns, as a quadword value, the default privileges for the user.

UAI\$_DFWSCNT

When UAI\$_DFWSCNT is specified, \$GETUAI returns the default working set size, which is a longword integer in the range 1 through 65,535.

UAI\$_DIOLM

When UAI\$_DIOLM is specified, \$GETUAI returns the direct I/O count limit, which is a longword integer in the range 1 through 65,535.

UAI\$_DIALUP_ACCESS_P

When UAI\$_DIALUP_ACCESS_P is specified, \$GETUAI returns the range of times during which dialup access is permitted for primary days. Each bit set in the longword represents a one-hour period, from bit 0 as midnight to 1 AM to bit 23 as 11 PM to midnight.

System Service Descriptions

\$GETUAI

UAIS_DIALUP_ACCESS_S

When UAIS_DIALUP_ACCESS_S is specified, \$GETUAI returns the range of times during which dialup access is permitted for secondary days. Each bit set in the longword represents a one-hour period, from bit 0 as midnight to 1 AM to bit 23 as 11 PM to midnight.

UAIS_ENCRYPT

When UAIS_ENCRYPT is specified, \$GETUAI returns, as a longword decimal value, a code indicating the encryption algorithm for the primary password.

UAIS_ENCRYPT2

When UAIS_ENCRYPT2 is specified, \$GETUAI returns, as a longword decimal value, a code indicating the encryption algorithm for the secondary password.

UAIS_ENQLM

When UAIS_ENQLM is specified, \$GETUAI returns the lock queue limit, which is a longword integer in the range 1 through 65,535.

UAIS_EXPIRATION

When UAIS_EXPIRATION is specified, \$GETUAI returns, as a quadword absolute time value, the expiration date and time of the account.

UAIS_FILLM

When UAIS_FILLM is specified, \$GETUAI returns the open file limit, which is a longword integer in the range 1 through 65,535.

UAIS_FLAGS

When UAIS_FLAGS is specified, \$GETUAI returns, as a longword bit vector, the various login flags set for the user. Each flag is represented by a bit. Listed below are the symbolic names for these flags, which are defined by the \$UAIDEF macro.

Symbol	Description
UAISV_AUDIT	All actions are audited.
UAISV_CAPTIVE	User is restricted to captive account.
UAISV_DEFCLI	User is restricted to default command interpreter.
UAISV_DISCTLY	User cannot use CTRL/Y.
UAISV_DISMAIL	Announcement of new mail is suppressed.
UAISV_DISRECONNECT	User cannot reconnect to existing processes.
UAISV_DISREPORT	User will not receive last login messages.
UAISV_DISWELCOME	User will not receive the login welcome message.
UAISV_GENPWD	User is required to use generated passwords.
UAISV_LOCKPWD	SET PASSWORD command is disabled.
UAISV_NOMAIL	Mail delivery to user is disabled.
UAISV_PWD_EXPIRED	Primary password is expired.
UAISV_PWD2_EXPIRED	Secondary password is expired.

UAIS_JTQUOTA

When UAIS_JTQUOTA is specified, \$GETUAI returns the initial byte quota with which the jobwide logical name table is to be created, which is a longword integer in the range 1 through 65,365.

System Service Descriptions

\$GETUAI

UAI\$_LASTLOGIN_I

When UAI\$_LASTLOGIN_I is specified, \$GETUAI returns, as a quadword absolute time value, the date of the last interactive login.

UAI\$_LASTLOGIN_N

When UAI\$_LASTLOGIN_N is specified, \$GETUAI returns, as a quadword absolute time value, the date of the last noninteractive login.

UAI\$_LGICMD

When UAI\$_LGICMD is specified, \$GETUAI returns, as an RMS file specification, the name of the default login command file. Since a file specification can include up to 255 characters plus a size-byte prefix, the buffer length field of the item descriptor should specify 256 (bytes).

UAI\$_LOCAL_ACCESS_P

When UAI\$_LOCAL_ACCESS_P is specified, \$GETUAI returns the range of times during which local interactive access is permitted for primary days. Each bit set in the longword represents a one-hour period, from bit 0 as midnight to 1 AM to bit 23 as 11 PM to midnight.

UAI\$_LOCAL_ACCESS_S

When UAI\$_LOCAL_ACCESS_S is specified, \$GETUAI returns the range of times during which batch access is permitted for secondary days. Each bit set in the longword represents a one-hour period, from bit 0 as midnight to 1 AM to bit 23 as 11 PM to midnight.

UAI\$_LOGFAILS

When UAI\$_LOGFAILS is specified, \$GETUAI returns the count of login failures, which is a longword integer in the range 1 through 65,535.

UAI\$_MAXACCTJOBS

When UAI\$_MAXACCTJOBS is specified, \$GETUAI returns, as a longword integer, the maximum number of batch, interactive, and detached processes which may be active at one time for all users of the same account. A value of 0 represents an unlimited number.

UAI\$_MAXDETACH

When UAI\$_MAXDETACH is specified, \$GETUAI returns the detached process limit, which is a longword integer in the range 1 through 65,535. A value of 0 represents an unlimited number.

UAI\$_MAXJOBS

When UAI\$_MAXJOBS is specified, \$GETUAI returns the active process limit, which is a longword integer in the range 1 through 65,535. A value of 0 represents an unlimited number.

UAI\$_NETWORK_ACCESS_P

When UAI\$_NETWORK_ACCESS_P is specified, \$GETUAI returns the range of times during which network access is permitted for primary days. Each bit set in the longword represents a one-hour period, from bit 0 as midnight to 1 AM to bit 23 as 11 PM to midnight.

UAI\$_NETWORK_ACCESS_S

When UAI\$_NETWORK_ACCESS_S is specified, \$GETUAI returns the range of times during which network access is permitted for secondary days. Each bit set in the longword represents a one-hour period, from bit 0 as midnight to 1 AM to bit 23 as 11 PM to midnight.

System Service Descriptions

\$GETUAI

UAI\$_OWNER

When UAI\$_OWNER is specified, \$GETUAI returns, as a character string, the name of the owner of the account. Since the owner name can include up to 31 characters plus a size-byte prefix, the buffer length field of the item descriptor should specify 32 (bytes).

UAI\$_PBYTLM

When UAI\$_PBYTLM is specified, \$GETUAI returns the paged buffer I/O byte count limit, which is a longword integer in the range 1 through 65,535.

UAI\$_PGFLQUOTA

When UAI\$_PGFLQUOTA is specified, \$GETUAI returns the paging file quota, which is a longword integer in the range 1 through 65,535.

UAI\$_PRCCNT

When UAI\$_PRCCNT is specified, \$GETUAI returns the subprocess creation limit, which is a longword integer in the range 1 through 65,535.

UAI\$_PRI

When UAI\$_PRI is specified, \$GETUAI returns the default base priority, which is a longword integer in the range 0 through 31.

UAI\$_PRIMEDAYS

When UAI\$_PRIMEDAYS is specified, \$GETUAI returns, as a longword bit vector, the primary and secondary days of the week. Each bit represents a day of the week, with the bit clear representing a primary day and the bit set representing a secondary day. Listed below are the symbolic names for these bits, which are defined by the \$UAIDEF macro.

UAI\$V_MONDAY
UAI\$V_TUESDAY
UAI\$V_WEDNESDAY
UAI\$V_THURSDAY
UAI\$V_FRIDAY
UAI\$V_SATURDAY
UAI\$V_SUNDAY

UAI\$_PRIV

When UAI\$_PRIV is specified, \$GETUAI returns, as a quadword value, the names of the privileges held by the user.

UAI\$_PWD

When UAI\$_PWD is specified, \$GETUAI returns, as a quadword value, the hashed primary password of the user.

UAI\$_PWD_DATE

When UAI\$_PWD_DATE is specified, \$GETUAI returns, as a quadword absolute time value, the date of the last password change.

UAI\$_PWD_LENGTH

When UAI\$_PWD_LENGTH is specified, \$GETUAI returns the minimum password length, which is a longword integer.

UAI\$_PWD_LIFETIME

When UAI\$_PWD_LIFETIME is specified, \$GETUAI returns, as a quadword absolute time value, the password lifetime.

System Service Descriptions

\$GETUAI

UAI\$_PWD2

When UAI\$_PWD2 is specified, \$GETUAI returns, as a quadword value, the hashed secondary password of the user.

UAI\$_PWD2_DATE

When UAI\$_PWD2_DATE is specified, \$GETUAI returns, as a quadword absolute time value, the last date the secondary password was changed.

UAI\$_QUEPRI

When UAI\$_QUEPRI is specified, \$GETUAI returns the maximum job queue priority, which is a longword integer in the range 0 through 31.

UAI\$_REMOTE_ACCESS_P

When UAI\$_REMOTE_ACCESS_P is specified, \$GETUAI returns the range of times during which remote interactive access is permitted for primary days. Each bit set in the longword represents a one-hour period, from bit 0 as midnight to 1 AM to bit 23 as 11 PM to midnight.

UAI\$_REMOTE_ACCESS_S

When UAI\$_REMOTE_ACCESS_S is specified, \$GETUAI returns the range of times during which remote interactive access is permitted for secondary days. Each bit set in the longword represents a one-hour period, from bit 0 as midnight to 1 AM to bit 23 as 11 PM to midnight.

UAI\$_SALT

When UAI\$_SALT is specified, \$GETUAI returns the random password salt, which is a longword integer in the range 1 through 65,535.

UAI\$_SHRFILLM

When UAI\$_SHRFILLM is specified, \$GETUAI returns the shared file limit, which is a longword integer in the range 1 through 65,535.

UAI\$_TQCNT

When UAI\$_TQCNT is specified, \$GETUAI returns the timer queue entry limit, which is a longword integer in the range 1 through 65,535.

UAI\$_UIC

When UAI\$_UIC is specified, \$GETUAI returns, as a longword, the user identification code (UIC), containing the following two word-length subfields:

Symbolic Name	Description
UIC\$W_MEM	The member number subfield of the UIC
UIC\$W_GRP	The group number subfield of the UIC

UAI\$_USERNAME

When UAI\$_USERNAME is specified, \$GETUAI returns the username of the owner of the specified job. The username is returned as a 12-byte, blank-filled string.

UAI\$_WSEXTENT

When UAI\$_WSEXTENT is specified, \$GETUAI returns the working set extent specified for the specified job or queue as a longword integer in the range 1 through 65,535.

System Service Descriptions

\$GETUAI

UAI\$_WSQUOTA

When UAI\$_WSQUOTA is specified, \$GETUAI returns the working set quota for the specified user as a longword integer in the range 1 through 65,535.

DESCRIPTION

Use the following list to determine the privileges required to use the \$GETUAI service:

- BYPASS or SYSPRV—allows access to any record in the UAF (user authorization file)
- GRPPRV—allows access to any record in the UAF whose UIC group matches that of the requester
- No privilege—allows access to any UAF record whose UIC matches that of the requester

CONDITION VALUES RETURNED

SS\$_NORMAL

Successful completion.

SS\$_ACCVIO

The item list or input buffer cannot be read by the caller; or the return length buffer, output buffer, or status block cannot be written by the caller.

SS\$_BADPARAM

The function code is invalid; the item list contains an invalid item code; a buffer descriptor has an invalid length; or the reserved parameter has a nonzero value.

SS\$_NOPRIV

The user does not have the privileges required to examine the authorization information for the specified user.

\$GRANTID

The Grant Identifier to Process service adds the specified identifier record to the process' rights list or to the system rights list. If the identifier is already in the rights list, the attributes are modified as specified.

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED".

```
VMS Usage:  rights_holder
type:       quadword (unsigned)
access:     modify
```

System Service Descriptions

\$GRANTID

mechanism: **by reference**

Identifier and attributes to be granted when \$GRANTID completes execution. The **id** argument is the address of a quadword containing the binary identifier code to be granted in the first longword and the attributes in the second longword.

Symbol values are offsets to the bits within the longword. You can also obtain the values as masks with the appropriate bit set using the prefix KGB\$M rather than KGB\$V. The symbols are defined in the system macro library (\$KGBDEF).

Bit	Meaning When Set
KGB\$V_DYNAMIC	Allows the unprivileged holder to add or remove the identifier from the process rights list
KGB\$V_RESOURCE	Allows the holder to charge resources, such as disk blocks, to the identifier

Either **id** or **name** must be specified. Since the **id** is returned if you specify **name**, as well as passed, you must pass it as a variable rather than a constant in this case.

name

VMS Usage: **char_string**

type: **character-coded text string**

access: **read only**

mechanism: **by descriptor-fixed length string descriptor**

Name of the identifier granted when \$GRANTID completes execution. The **name** argument is the address of a descriptor pointing to the name of the identifier. Either **id** or **name** must be specified.

prvatr

VMS Usage: **mask_longword**

type: **longword (unsigned)**

access: **write only**

mechanism: **by reference**

Previous attributes of the identifier. The **prvatr** argument is the address of a longword used to store the attributes of the identifier if it was previously present in the rights list. If the identifier was added rather than modified, **prvatr** is ignored.

DESCRIPTION

The Grant Identifier to Process service adds the specified identifier to the process's rights list, or to the system rights list. If the identifier is already in the rights list, its attributes are modified to those specified. This service is meant for use by a privileged subsystem to alter the access rights profile of a user, based on installation policy. It is not meant for use by the general system user.

CMKRNL privilege is required to invoke this service. In addition, GROUP privilege is required to modify the rights list of a process in the same group as the calling process (unless the process has the same UIC as the calling process). WORLD privilege is required to modify the rights list of a process outside the caller's group. SYSNAM privilege is required to modify the system rights list.

System Service Descriptions

\$GRANTID

The result of passing the **pidadr** or the **prcnam** argument or both to SYS\$GRANTID is summarized in the following table:

prcnam	pidadr	Result
Omitted	Omitted	Current process id is used; process id is not returned.
Omitted	0	Current process id is used; process id is returned.
Omitted	Specified	Specified process id is used; process id is returned.
Specified	Omitted	Specified process name is used; process id is not returned.
Specified	0	Specified process name is used; process id is returned.
Specified	Specified	Specified process id is used; process id is returned; process name is ignored.

The result of passing either the **name** or the **id** argument or both to SYS\$GRANTID is summarized in the following table:

name	id	Result
Omitted	Omitted	Illegal.
Omitted	Specified	Specified identifier value is used; identifier value is returned.
Specified	Omitted	Specified identifier name is used; identifier value is not returned.
Specified	0	Specified identifier name is used; identifier value is returned.
Specified	Specified	Specified identifier value is used; identifier value is returned; identifier name is ignored.

CONDITION VALUES RETURNED

SS\$_WASCLR	Service is successfully completed; the rights list did not contain the specified identifier.
SS\$_WASSET	Service is successfully completed; the rights list already held the specified identifier.
SS\$_ACCVIO	The pidadr cannot be read or written, or prcnam cannot be read, or id cannot be read or written, or the name cannot be read, or prvatr cannot be written.
SS\$_IIDENT	The specified identifier or holder is of invalid format, or the specified identifier and holder are equal.
SS\$_INSFMEM	Insufficient process dynamic memory is available to open the rights database.
SS\$_NOPRIV	The caller does not have CMKRNL privilege, or is not running in exec or kernel mode, or the caller lacks GROUP, WORLD, or SYSNAM privilege as required.

System Service Descriptions

\$GRANTID

SS\$_NOSUCHID

The specified identifier name does not exist in the rights database. Note that the binary identifier, if given, is not validated against the rights database.

SS\$_RIGHTSFULL

The process's or system rights list is full.

SS\$_NOSYSNAM

The operation requires SYSNAM privilege.

SS\$_IVLOGNAM

Invalid logical name has been specified.

SS\$_NONEXPR

Nonexistent process has been specified.

RMS\$_PRV

The user does not have read access to the rights database.

Since the rights database is an indexed file accessed with VAX RMS, this service may also return RMS status codes associated with operations on indexed files. Please refer to the *VAX Record Management Services Reference Manual* for descriptions of these status codes.

System Service Descriptions

\$HIBER

\$HIBER—Hibernate

The Hibernate service allows a process to make itself inactive but to remain known to the system so that it can be interrupted, for example, to receive ASTs. A hibernate request is a wait-for-wake-event request. When the Wake Process from Hibernation (\$WAKE) service is called or when a Schedule Wakeup (\$SCHDWK) service comes due, the process continues execution at the instruction following the Hibernate call.

FORMAT

SY\$HIBER

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS

None.

DESCRIPTION

In VAX MACRO, you can call the Hibernate service only by using the \$name_S macro.

A hibernating process can be swapped out of the balance set if it is not locked into the balance set.

The wait state caused by \$HIBER can be interrupted by an AST if the access mode at which the AST is to execute is equal to or more privileged than the access mode from which the hibernate request was issued and the process is enabled for ASTs at that access mode.

When the AST service routine completes execution, the system reexecutes the \$HIBER service on the process's behalf. If a wakeup request has been issued for the process during the execution of the AST service routine (either by itself or another process), the process resumes execution. If a wakeup request has not been issued, it continues to hibernate.

If one or more wakeup requests are issued for the process while it is not hibernating, the next hibernate call returns immediately, that is, the process does not hibernate. No count is maintained of outstanding wake-up requests.

Although this service has no arguments, a FORTRAN function reference must use parentheses to indicate a null argument list, as in the following example:

```
ISTAT=SYS$HIBER()
```

System Service Descriptions

\$HIBER

**CONDITION
VALUES
RETURNED**

SS\$_NORMAL

Service successfully completed.

\$IDTOASC

\$IDTOASC—Translate Identifier to Identifier Name

The Translate Identifier to Identifier Name service translates the specified identifier value to its identifier name.

FORMAT

SY\$IDTOASC *id* ,*[namlen]* ,*[nambuf]* ,*[resid]* ,*[attrib]*
,[contxt]

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS *id*

VMS Usage: **rights_id**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Binary identifier value translated by \$IDTOASC. The **id** argument is a longword containing the binary value of the identifier. To determine the identifier names of all identifiers in the rights database, specify **id** as **-1** and call SY\$IDTOASC repeatedly until it returns the status code SS\$_NOSUCHID. The identifiers are returned in alphabetical order.

namlen

VMS Usage: **word_unsigned**
type: **word (unsigned)**
access: **write only**
mechanism: **by reference**

Number of characters in the identifier name translated by \$IDTOASC. The **namlen** argument is the address of a word containing the length of the identifier name written to **nambuf**.

nambuf

VMS Usage: **char_string**
type: **character-coded text string**
access: **write only**
mechanism: **by descriptor—fixed length string descriptor**

Identifier name text string returned when \$IDTOASC completes the translation. The **nambuf** argument is the address of a descriptor pointing to the buffer in which the identifier name is written.

resid

VMS Usage: **rights_id**
 type: **longword (unsigned)**
 access: **write only**
 mechanism: **by reference**

Identifier value of the identifier name returned in **nambuf**. The **resid** argument is the address of a longword containing the 32-bit code of the identifier.

attrib

VMS Usage: **mask_longword**
 type: **longword (unsigned)**
 access: **write only**
 mechanism: **by reference**

Mask of attributes associated with the identifier returned in **resid**. The **attrib** argument is the address of a longword containing the attribute mask.

Symbol values are offsets to the bits within the longword. You can also obtain the values as masks with the appropriate bit set using the prefix **KGB\$M** rather than **KGB\$V**. The symbols are defined in the system macro library (**\$KGBDEF**).

Bit	Meaning When Set
KGB\$_DYNAMIC	Allows the unprivileged holder to add or remove the identifier from the process rights list
KGB\$V_RESOURCE	Allows the holder to charge resources, such as disk blocks, to the identifier

contxt

VMS Usage: **context**
 type: **longword (unsigned)**
 access: **modify**
 mechanism: **by reference**

Context value used when repeatedly calling **\$IDTOASC**. The **contxt** argument is the address of a longword used while searching for all identifiers. The context value must be initialized to zero, and the resulting context of each call to **\$IDTOASC** must be presented to each subsequent call. Once **contxt** has been passed to **\$IDTOASC**, do not modify its value.

DESCRIPTION

The Translate Identifier to Identifier Name service translates the specified binary identifier value to an identifier name. While the primary purpose of the Translate Identifier to Identifier Name service is to translate the specified identifier to its name, it may also be used to find all identifiers in the rights database. To determine all the identifiers, call **\$IDTOASC** repeatedly until it returns the status code **SS\$_NOSUCHID**. When **SS\$_NOSUCHID** is returned, **\$IDTOASC** has returned all the identifiers, cleared the context value, and deallocated the record stream.

If you complete your calls to **\$IDTOASC** before **SS\$_NOSUCHID** is returned, use **SYS\$FINISH_RDB** to clear the context value and deallocate the record stream.

When you use wildcards with this service, the records are returned in identifier name order.

System Service Descriptions

\$IDTOASC

CONDITION VALUES RETURNED

SS\$_NORMAL	Service is successfully completed.
SS\$_ACCVIO	The namlen , nambuf , resid , attrib , or ctxt arguments cannot be written by the caller.
SS\$_INSFMEM	Insufficient process dynamic memory is available to open the rights database.
SS\$_IVCHAN	The contents of the context longword are not valid.
SS\$_IVIDENT	The specified identifier is of invalid format.
SS\$_NOIOCHAN	No more rights database context streams are available.
SS\$_NOSUCHID	The specified identifier name does not exist in the rights database, or the entire rights database has been searched if the id is -1.
RMS\$_PRV	The user does not have read access to the rights database.

Since the rights database is an indexed file accessed with VAX RMS, this service may also return RMS status codes associated with operations on indexed files. Please refer to the *VAX Record Management Services Reference Manual* for descriptions of these status codes.

\$LCKPAG—Lock Pages in Memory

The Lock Pages In Memory service locks a page or range of pages in memory. The specified virtual pages are forced into the working set and then locked in memory. A locked page is not swapped out of memory if its process's working set is swapped out. These pages are not candidates for page replacement and in this sense are locked in the working set as well.

FORMAT **SYSLCKPAG** *inadr* , [*retadr*] , [*acmode*]

RETURNS

VMS Usage: **cond_value**
 type: **longword (unsigned)**
 access: **write only**
 mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS

inadr

VMS Usage: **address_range**
 type: **longword (unsigned)**
 access: **read only**
 mechanism: **by reference**

Starting and ending virtual addresses of the range of pages to be locked. The *inadr* is the address of a two-longword array containing, in order, the starting and the ending process virtual addresses. Only the virtual page number portion of each virtual address is used; the low-order 9 bits are ignored.

If the starting and ending virtual addresses are the same, a single page is locked.

retadr

VMS Usage: **address_range**
 type: **longword (unsigned)**
 access: **write only**
 mechanism: **by reference—array reference or descriptor**

Starting and ending process virtual addresses of the pages that \$LCKPAG actually locked. The *retadr* is the address of a two-longword array containing, in order, the starting and ending process virtual addresses.

acmode

VMS Usage: **access_mode**
 type: **longword (unsigned)**
 access: **read only**
 mechanism: **by value**

Access mode to be associated with the pages to be locked. The *acmode* argument is a longword containing the access mode. The \$PSLDEF macro defines the four access modes.

System Service Descriptions

\$LCKPAG

The most privileged access mode used is the access mode of the caller. For the \$LCKPAG service to complete successfully, the resultant access mode must be equal to or more privileged than the access mode already associated with the pages to be locked.

DESCRIPTION

The calling process must have PSWAPM privilege to lock pages into memory. If more than one page is being locked and it is necessary to determine specifically which pages had been previously locked, the pages should be locked one at a time.

If an error occurs while locking pages, the return array, if requested, indicates the pages that were successfully locked before the error occurred. If no pages are locked, both longwords in the return address array contain the value -1.

Pages that are locked in memory can be unlocked with the Unlock Pages from Memory (\$ULKPAG) service. Locked pages are automatically unlocked at image exit.

**CONDITION
VALUES
RETURNED**

SS\$_WASCLR	Service successfully completed. All of the specified pages were previously unlocked.
SS\$_WASSET	Service successfully completed. At least one of the specified pages was previously locked.
SS\$_ACCVIO	The input array cannot be read by the caller; the output array cannot be written by the caller; or a page in the specified range is inaccessible or does not exist.
SS\$_LCKPAGFUL	The system-defined maximum limit on the number of pages that can be locked in memory has been reached.
SS\$_NOPRIV	The process does not have the privilege to lock pages in memory.

\$LKWSET—Lock Pages in Working Set

The Lock Pages in Working Set service locks a range of pages in the working set; if the pages are not already in the working set, it brings them in and locks them. A page that is locked in the working set does not become a candidate for replacement.

FORMAT **SYSLKWSET** *inadr* , [*retadr*] , [*acmode*]

RETURNS

VMS Usage: **cond_value**
 type: **longword (unsigned)**
 access: **write only**
 mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS *inadr*

VMS Usage: **address_range**
 type: **longword (unsigned)**
 access: **read only**
 mechanism: **by reference**

Starting and ending virtual addresses of the range of pages to be locked in the working set. The *inadr* is the address of a two-longword array containing, in order, the starting and the ending process virtual addresses. Only the virtual page number portion of each virtual address is used; the low-order 9 bits are ignored.

If the starting and ending virtual addresses are the same, a single page is locked.

retadr

VMS Usage: **address_range**
 type: **longword (unsigned)**
 access: **write only**
 mechanism: **by reference**

Starting and ending process virtual addresses of the range of pages actually locked by \$LKWSET. The *retadr* is the address of a two-longword array containing, in order, the starting and ending process virtual addresses.

acmode

VMS Usage: **access_mode**
 type: **longword (unsigned)**
 access: **read only**
 mechanism: **by value**

Access mode to be associated with the pages to be locked. The *acmode* argument is a longword containing the access mode. The \$PSLDEF macro defines the four access modes.

System Service Descriptions

\$LKWSET

The most privileged access mode used is the access mode of the caller. For the \$LKWSET service to complete successfully, the resultant access mode must be equal to or more privileged than the access mode already associated with the pages to be locked.

DESCRIPTION If more than one page is being locked and it is necessary to determine specifically which pages had been previously locked, the pages should be locked one at a time.

If an error occurs while locking pages, the return array, if requested, indicates the pages that were successfully locked before the error occurred. If no pages are locked, both longwords in the return address array contain a -1.

Pages that are locked in the working set can be unlocked with the Unlock Page from Working Set (\$ULWSET) service.

Global pages with write access cannot be locked into the working set.

CONDITION VALUES RETURNED	SS\$_WASCLR	Service successfully completed. All of the specified pages were previously unlocked.
	SS\$_WASSET	Service successfully completed. At least one of the specified pages was previously locked in the working set.
	SS\$_ACCVIO	The input address array cannot be read by the caller; the output address array cannot be written by the caller; or a page in the specified range is inaccessible or nonexistent.
	SS\$_LKWSETFUL	The locked working set is full. If any more pages are locked, there will not be enough dynamic pages available to continue execution.
	SS\$_NOPRIV	A page in the specified range is in the system address space, or a global page with write access was specified.
	SS\$_PAGOWNVIO	The pages could not be locked because the access mode associated with the call to \$LKWSET was less privileged than the access mode associated with the pages that were to be locked.

\$MGBLSC—Map Global Section

The Map Global Section service establishes a correspondence between (maps) pages in the process's virtual address space and physical pages occupied by a global section.

FORMAT **SYS\$MGBLSC** *inadr,[retadr],[acmode],[flags]*
 ,gsdnam,[ident],[relpag]

RETURNS

VMS Usage: cond_value
type: longword (unsigned)
access: write only
mechanism: by value

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS *inadr*

VMS Usage: **address_range**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Starting and ending virtual addresses in the process's virtual address space (either the P0 or P1 regions) into which the section is to be mapped. The **inadr** is the address of a two-longword array containing, in order, the starting and the ending process virtual addresses. Only the virtual page number portion of each virtual address is used; the low-order 9 bits are ignored.

If the starting and ending virtual addresses are the same, a single page is mapped (except when the SEC\$M_EXPREG bit is set in the **flags** argument).

If the `SEC$M_EXPREG` bit is set in the `flags` argument, the starting address (first longword) specified in the `inadr` argument simply determines whether the section is mapped in the program (P0) region or control (P1) region; the ending address (second longword) is ignored.

retadr

VMS Usage: **address_range**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Starting and ending process virtual addresses into which the section was actually mapped by \$MGBLSC. The **retadr** is the address of a two-longword array containing, in order, the starting and ending process virtual addresses.

acmode

VMS Usage: **access_mode**
type: **longword (unsigned)**
access: **read only**

System Service Descriptions

\$MGBLSC

mechanism: **by value**

Access mode to be associated with the pages that are mapped into the process virtual address space. The **acmode** argument is a longword containing the access mode. The \$PSLDEF macro defines symbols for the four access modes.

The most privileged access mode used is the access mode of the caller.

flags

VMS Usage: **mask_longword**

type: **longword (unsigned)**

access: **read only**

mechanism: **by value**

Flag mask specifying options for the operation. The **flags** argument is a longword bit vector wherein a bit when set specifies the corresponding option.

Symbolic names for the flag bits are defined by the \$SECDEF macro. The **flags** argument is constructed by specifying the symbolic names of each desired option in a logical OR operation. The following list describes each flag option:

Flag	Description
SEC\$_WRT	Map section with read/write access. By default, the section is mapped with read-only access.
SEC\$_SYSGBL	Map a system global section. By default, the section is a group global section.
SEC\$_EXPREG	Map the section in the first available virtual address range. By default, the section is mapped into the range specified by the inadr argument.

gsdnam

VMS Usage: **section_name**

type: **character-coded text string**

access: **read only**

mechanism: **by descriptor—fixed length string descriptor**

Name of the global section. The **gsdnam** argument is the address of a character string descriptor pointing to this name string. Section 11.6.5.1 describes the format of this name string.

For group global sections, VAX/VMS interprets the group UIC as part of the global section name; thus, the names of global sections are unique to UIC groups. Further, all global section names are implicitly qualified by their identification fields.

ident

VMS Usage: **section_id**

type: **quadword (unsigned)**

access: **read only**

mechanism: **by reference**

Identification value specifying the version number of a global section, and; for processes mapping to an existing global section, the criteria for matching the identification. The **ident** argument is the address of a quadword structure containing three fields.

System Service Descriptions

\$MGBLSC

The first longword specifies, in the low-order 3 bits, the matching criteria. Their valid values, the symbolic names by which they can be specified, and their meanings are listed below.

Value/Name	Match Criteria
0 SEC\$_MATALL	Match all versions of the section.
1 SEC\$_MATEQU	Match only if major and minor identifications match.
2 SEC\$_MATLEQ	Match if the major identifications are equal and the minor identification of the mapper is less than or equal to the minor identification of the global section.

The version number is in the second longword. The version number contains two fields: a minor identification in the low-order 24 bits and a major identification in the high-order 8 bits.

If **ident** is not specified or is specified as 0 (the default), the version number and match control fields default to 0.

relpag

VMS Usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Relative page number within the section of the first page to be mapped. The **relpag** argument is a longword containing this number.

If **relpag** is not specified or is specified as 0 (the default), the global section is mapped beginning with the first virtual block in the section.

DESCRIPTION

The protection mask specified at the time the global section is created determines the type of access (for example, read/write or read/only) that a particular process has to the section.

\$MGBLS uses the following system resources:

- The process's working set limit quota (WSQUOTA) must be sufficient to accommodate the increased size of the virtual address space when mapping a section.
- If the section pages are copy-on-reference, the process must also have sufficient paging file quota (PGFLQUOTA)

Use of this system service causes the calling process's working set to be adjusted to the size specified by the working set quota (WSQUOTA). If the process's working set size is less than quota, the working set size is increased; if the process's working set size is greater than quota, the working set size is decreased.

When \$MGBLSC maps a global section, it adds pages to the process's virtual address space. The section is mapped from a low address to a high address, regardless of whether the section is mapped in the program or control region.

If an error occurs during the mapping of a global section, the return address array, if specified, indicates the pages that were successfully mapped when the error occurred. If no pages were mapped, both longwords of the return address array contain -1.

System Service Descriptions

\$MGBLSC

CONDITION VALUES RETURNED

SS\$_NORMAL

Service successfully completed.

SS\$_ACCVIO

The input address array, the global section name or name descriptor, or the section identification field cannot be read by the caller; or the return address array cannot be written by the caller.

SS\$_ENDOFFILE

Warning. The starting virtual block number specified is beyond the logical end-of-file.

SS\$_EXQUOTA

The process exceeded its paging file quota, creating copy-on-reference pages.

SS\$_INSFWSL

The process's working set limit is not large enough to accommodate the increased virtual address space.

SS\$_INTERLOCK

The bit map lock for allocating global sections from the specified shared memory is locked by another process.

SS\$_JVLOGNAM

The global section name has a length of 0 or has more than 15 characters.

SS\$_JVSECFLG

A reserved flag was set.

SS\$_JVSECIDCTL

The match control field of the global section identification is invalid.

SS\$_NOPRIV

The file protection mask specified when the global section was created prohibits the type of access requested by the caller; or a page in the input address range is in the system address space.

SS\$_NOSUCHSEC

Warning. The specified global section does not exist.

SS\$_PAGOWNVIO

A page in the specified input address range is owned by a more privileged access mode.

SS\$_SHMNOTCNCT

The shared memory named in the **gsdnam** argument is not known to the system. This error can be caused by a spelling error in the string, an improperly assigned logical name, or the failure to identify the memory as shared at system generation time.

SS\$_TOOMANYLNAM

Logical name translation of the **gsdnam** string exceeded the allowed depth.

SS\$_VASFULL

The process's virtual address space is full; no space is available in the page tables for the pages created to contain the mapped global section.

\$MOD_HOLDER—Modify Holder Record in Rights Database

The Modify Holder Record in Rights Database service modifies the specified holder record of the target identifier in the rights database. Identifier attributes may be added, or removed, or both.

FORMAT **SY\$MOD_HOLDER** *id* ,holder ,[set_attr] ,[clr_attr]

RETURNS

VMS Usage: **cond_value**
 type: **longword (unsigned)**
 access: **write only**
 mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED".

ARGUMENTS *id*

VMS Usage: **rights_id**
 type: **longword (unsigned)**
 access: **read only**
 mechanism: **by value**

Binary value of target identifier whose holder record is modified when \$MOD_HOLDER completes execution. The **id** argument is a longword containing the identifier value.

holder

VMS Usage: **rights_holder**
 type: **quadword (unsigned)**
 access: **read only**
 mechanism: **by reference**

Identifier of holder being modified when \$MOD_HOLDER completes execution. The **holder** argument is the address of a quadword containing the UIC identifier of the holder in the first longword and the value of zero in the second longword.

set_attr

VMS Usage: **mask_longword**
 type: **longword (unsigned)**
 access: **read only**
 mechanism: **by value**

Bitmask of attributes to be enabled for the identifier when \$MOD_HOLDER completes execution. The **set_attr** argument is a longword containing the attribute mask.

System Service Descriptions

\$MOD_HOLDER

The attributes actually enabled are the intersection of those specified and the attributes of the identifier. If the same attribute is specified in **set_attr** and **clr_attr**, the attribute is enabled.

Symbol values are offsets to the bits within the longword. You can also obtain the values as masks with the appropriate bit set using the prefix **KGB\$M** rather than **KGB\$V**. The symbols are defined in the system macro library (**\$KGBDEF**).

Bit	Meaning When Set
KGB\$V_DYNAMIC	Allows the unprivileged holder to add or remove the identifier from the process rights list.
KGB\$V_RESOURCE	Allows the holder to charge resources, such as disk blocks, to the identifier.

clr_attr

VMS Usage: **mask_longword**

type: **longword (unsigned)**

access: **read only**

mechanism: **by value**

Bitmask of attributes to be disabled for the identifier when **\$MOD_HOLDER** completes execution. The **clr_attr** argument is a longword containing the attribute mask.

If the same attribute is specified in **set_attr** and **clr_attr**, the attribute is enabled.

Symbol values are offsets to the bits within the longword. You can also obtain the values as masks with the appropriate bit set using the prefix **KGB\$M** rather than **KGB\$V**. The symbols are defined in the system macro library (**\$KGBDEF**).

Bit	Meaning When Set
KGB\$V_DYNAMIC	Allows the unprivileged holder to add or remove the identifier from the process rights list.
KGB\$V_RESOURCE	Allows the holder to charge resources, such as disk blocks, to the identifier.

DESCRIPTION

The Modify Holder Record In Rights Database service modifies the specified holder record in the rights database. Identifier attributes may be added, or removed, or both.

When you specify both the **set_attr** and **clr_attr** arguments, the attribute is cleared first. Thus, if you were to specify the same attribute bit with each argument, the result would be that the bit would be set.

Write access to the rights database is required to use this service. If the database is in **SYS\$SYSTEM** (which is the default) **SYSPRV** privilege is needed to grant write access to the database.

System Service Descriptions

\$MOD_HOLDER

CONDITION VALUES RETURNED

SS\$_NORMAL	Successful completion.
SS\$_ACCVIO	The holder argument cannot be read by the caller.
SS\$_BADPARAM	The specified attributes contain invalid attribute flags.
SS\$_INSFMEM	Insufficient process dynamic memory is available to open the rights database.
SS\$_IIDENT	The specified identifier or holder identifier is of invalid format.
SS\$_NOSUCHID	The specified identifier does not exist in the rights database, or the specified holder identifier does not exist in the rights database.
RMS\$_PRV	The user does not have write access to the rights database.

Since the rights database is an indexed file accessed with VAX RMS, this service may also return RMS status codes associated with operations on indexed files. Please refer to the *VAX Record Management Services Reference Manual* for descriptions of these status codes.

\$MOD_IDENT

The **Modify Identifier** in **Rights Database** service modifies the specified identifier record in the rights database. Identifier attributes may be added, or removed, or both. The identifier name or value may be changed.

RETURNS

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED".

VMS Usage: **rights_id**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Binary value of identifier whose identifier record is modified when \$MOD_IDENT completes execution. The **id** argument is a longword containing the identifier value.

VMS Usage: **mask_longword**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Bit mask of attributes to be enabled for the identifier when \$MOD_IDENT completes execution. The **set_attr** argument is a longword containing the attribute mask.

The attributes actually enabled are the intersection of those specified and the attributes of the identifier. If the same attribute is specified in `set_attrib` and `clr_attrib`, the attribute is enabled.

Symbol values are offsets to the bits within the longword. You can also obtain the values as masks with the appropriate bit set using the prefix `KGB$M` rather than `KGB$V`. The symbols are defined in the system macro library (`$KGBDEF`).

System Service Descriptions

\$MOD_IDENT

Bit	Meaning When Set
KGB\$V_DYNAMIC	Allows the unprivileged holder to add or remove the identifier from the process rights list.
KGB\$V_RESOURCE	Allows the holder to charge resources, such as disk blocks, to the identifier.

clr_attrib

VMS Usage: **mask_longword**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Bit mask of attributes to be disabled for the identifier when \$MOD_IDENT completes execution. The **clr_attrib** argument is a longword containing the attribute mask.

If the same attribute is specified in **set_attrib** and **clr_attrib**, the attribute is enabled.

Symbol values are offsets to the bits within the longword. You can also obtain the values as masks with the appropriate bit set using the prefix KGB\$M rather than KGB\$V. The symbols are defined in the system macro library (\$KGBDEF).

Bit	Meaning When Set
KGB\$V_DYNAMIC	Allows the unprivileged holder to add or remove the identifier from the process rights list.
KGB\$V_RESOURCE	Allows the holder to charge resources, such as disk blocks, to the identifier.

new_name

VMS Usage: **char_string**
type: **character-coded text string**
access: **read only**
mechanism: **by descriptor—fixed length string descriptor**

New name to be given to the specified identifier. The **new_name** argument is the address of the descriptor pointing to the identifier name string.

An identifier name consists of 1 to 31 alphanumeric characters including dollar signs and underscores, containing at least one nonnumeric character. Any lowercase characters specified are automatically converted to uppercase.

new_value

VMS Usage: **rights_id**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

New value to be assigned to the specified identifier. The **new_value** argument is a longword containing the binary value of the specified identifier. When the identifier value is changed, \$MOD_IDENT also changes the value of the identifier in all of the holder records in which the specified identifier appears.

System Service Descriptions

\$MOD_IDENT

DESCRIPTION The Modify Identifier in Rights Database service modifies the specified identifier record in the rights database. When you specify both the **set_attr** and **clr_attr** arguments, the attribute is cleared first. Thus, if you were to specify the same attribute bit with each argument, the result would be that the bit would be set.

Write access to the rights database is required to use this service. If the database is in SYS\$SYSTEM (which is the default) SYSPRV privilege is needed to grant write access to the database.

**CONDITION
VALUES
RETURNED**

SS\$_NORMAL	Service is successfully completed.
SS\$_NOSUCHID	The specified identifier does not exist in the rights database.
SS\$_BADPARAM	The specified attributes contain invalid attribute flags.
SS\$_INSFMEM	Insufficient process dynamic memory is available to open the rights database.
SS\$_IVIDENT	The specified identifier is of invalid format.
RMS\$_PRV	The user does not have write access to the rights database.

Since the rights database is an indexed file accessed with VAX RMS, this service may also return RMS status codes associated with operations on indexed files. Please refer to the *VAX Record Management Services Reference Manual* for descriptions of these status codes.

\$MOUNT—Mount Volume

The Mount Volume service mounts a tape, disk volume, or volume set and specifies options for the mount operation.

FORMAT **SY\$MOUNT** *itmlst*

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENT

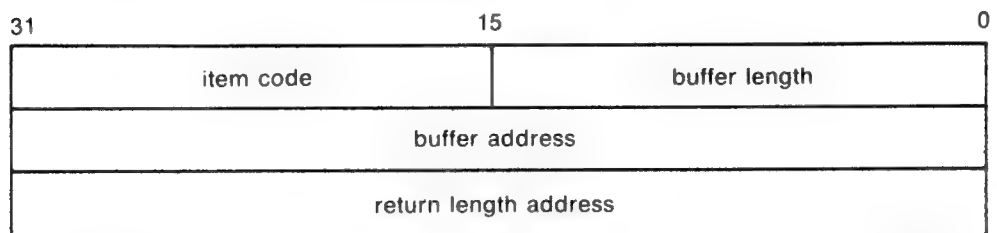
itmlst

VMS Usage: **item_list_3**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Item list specifying options for the mount operation. The ***itmlst*** argument is the address of a list of item descriptors, each of which specifies an option and provides the information needed to perform the operation.

The item list must include at least one device item descriptor. The item list is terminated by a longword of 0.

The following diagram depicts the format of a single item descriptor:



ZK-1705-84

\$MOUNT Item Descriptor Fields

buffer length

A word specifying the length (in bytes) of the buffer that supplies the information needed by \$MOUNT to process the specified item code. The required length of the buffer depends upon the item code specified in the **item code** field of the item descriptor. If the value of **buffer length** is too small, \$MOUNT truncates the data.

System Service Descriptions

\$MOUNT

item code

A word containing a user-supplied symbolic code that specifies an option for the mount operation. These codes are defined by the \$MNTDEF macro. Each item code is described in the "\$MOUNT Item Codes" section.

buffer address

A longword containing the address of the buffer that supplies information to \$MOUNT.

return length address

This field is not used.

\$MOUNT Item Codes

MNT\$_DEVNAM

MNT\$_DEVNAM specifies the name of the device to be mounted. The buffer must contain a character string of from 1 to 64 characters, which is the device name. The device name may be a physical device name or a logical name; if it is a logical name, it must translate to a physical device name.

MNT\$_DEVNAM must appear at least once in an item list, and it may appear more than once. It appears more than once when a volume set is being mounted, since, in this case, one device is being mounted for each volume in the volume set.

MNT\$_VOLNAM

MNT\$_VOLNAM specifies the name of the volume to be mounted on the device. The buffer must contain a character string of from 1 to 12 characters, which is the volume name.

MNT\$_VOLNAM may appear more than once in an item list; it appears more than once when a volume set is being mounted, since, in this case, one volume name is given to each volume in the volume set.

When a disk volume set is being mounted, the MNT\$_DEVNAM and MNT\$_VOLNAM item codes must each be specified once for each volume of the volume set. \$MOUNT mounts the volume specified by the first MNT\$_VOLNAM item code on the device specified by the first MNT\$_DEVNAM item code in the item list; it mounts the volume specified by the second MNT\$_VOLNAM code on the device specified by the second MNT\$_DEVNAM code; and so on for all specified volumes and devices. Thus, there must be an equal number of these two item codes in the item list.

When a tape volume set is being mounted, the number of MNT\$_DEVNAM item codes specified need not be equal to the number of MNT\$_VOLNAM item codes specified; this is true because more than one volume may be mounted on the same device.

MNT\$_LOGNAM

MNT\$_LOGNAM specifies a logical name for the volume; this logical name is equated to the device name specified by the first MNT\$_DEVNAM item code. The buffer must contain a character string of from 1 to 64 characters, which is the logical name.

Unless MNT\$M_GROUP or MNT\$M_SYSTEM is specified, the logical name is entered in the process logical name table.

System Service Descriptions

\$MOUNT

MNT\$_FLAGS

MNT\$_FLAGS specifies a longword bit vector wherein each bit specifies an option for the mount operation. The buffer must contain a longword, which is the bit vector.

The \$MNTDEF macro defines symbolic names for each option (bit) in the bit vector. The bit vector is constructed by specifying the symbolic names for the desired options in a logical OR operation. The following list gives the symbolic names for each option:

Option	Description
MNT\$_FOREIGN	The volume is to be mounted as a foreign volume; a foreign volume is not Files-11 structured. If MNT\$_FOREIGN is specified, the following item codes may each appear in the item list only once: MNT\$_DEVNAM, MNT\$_VOLNAM, and MNT\$_LOGNAM. To specify MNT\$_FOREIGN, the caller must either own the volume or have VOLPRO privilege.
MNT\$_GROUP	The logical name for the volume to be mounted is entered in the group logical name table, and the volume is made accessible to other users with the same UIC group number as that of the calling process. To specify MNT\$_GROUP, the caller must have GRPNAM privilege. MNT\$_GROUP applies only to disks.
MNT\$_NOASSIST	\$MOUNT does not request operator assistance if errors are encountered during the mount operation. If not specified, \$MOUNT requests operator assistance to recover from some error conditions.
MNT\$_NODISKQ	Disk quotas are not to be enforced for the volume to be mounted. If not specified, disk quotas are enforced. To specify MNT\$_NODISKQ, the caller must either own the volume or have VOLPRO privilege. MNT\$_NODISKQ applies only to disks.
MNT\$_NOHDR3	ANSI HDR3 and HDR4 labels are not to be written to magnetic tapes as they are mounted. If not specified, ANSI HDR3 and HDR4 labels are written to all tapes. Use MNT\$_NOHDR3 when writing to volumes that will be read by a system, such as the RT-11 system, which does not process HDR3 and HDR4 labels correctly. MNT\$_NOHDR3 applies only to tapes.
MNT\$_NOWRITE	The volume to be mounted is software write locked. If not specified, the volume is assumed to have read and write access.

System Service Descriptions

\$MOUNT

Option	Description
MNT\$M_OVR_ACCESS	<p>If the installation allows, this option overrides any character in the Accessibility Field of the volume. The necessity of this option is defined by the installation. That is, each installation has the option of specifying a routine that the magnetic tape file system will use to process this field. By default, VAX/VMS provides a routine that checks this field in the following manner:</p> <ul style="list-style-type: none">• If the magnetic tape was created on a version of VAX/VMS that conforms to Version 3 of ANSI, then this option must be used to override any character other than an ASCII space.• If a VAX/VMS protection is specified and that magnetic tape conforms to an ANSI standard that is later than Version 3, then this option must be used to override any character other than an ASCII 1. <p>To specify MNT\$M_OVR_ACCESS, the caller must either own the volume or have VOLPRO privilege. MNT\$M_OVR_ACCESS applies only to tapes.</p>
MNT\$M_OVR_EXP	<p>A tape that has not yet reached its expiration date may be overwritten. To specify MNT\$M_OVR_EXP, the caller must own the volume or have VOLPRO privilege.</p>
MNT\$M_OVR_IDENT	<p>The volume may be mounted without specifying the volume name (by using the MNT\$_VOLNAM item code). If specified, the following options must not be specified: MNT\$M_GROUP, MNT\$M_SHARE, and MNT\$M_SYSTEM.</p>
MNT\$M_OVR_LOCK	<p>The software write lock that occurs when a volume has a corrupted storage bit mask may be overridden.</p>
MNT\$M_OVR_SETID	<p>Checks on the volume set identification are not to be performed when subsequent reels in the volume set are mounted. MNT\$M_OVR_SETID applies only to tapes.</p>
MNT\$M_READCHECK	<p>Read checks are to be performed following all read operations.</p>
MNT\$M_SHARE	<p>Volume is to be mounted shared and therefore is to be accessible to other users. MNT\$M_SHARE applies only to disks.</p> <p>If the volume was previously mounted shared by another user and MNT\$M_SHARE is specified in the current call, all other options specified in the current call are ignored.</p> <p>If the caller allocated the device and specified MNT\$M_SHARE in the call to \$MOUNT, \$MOUNT will deallocate the device so that other users may access the volume.</p>
MNT\$M_MESSAGE	<p>Messages will be sent to the caller's SYS\$OUTPUT device.</p>

System Service Descriptions

\$MOUNT

Option	Description
MNT\$M_SYSTEM	The logical name for the volume to be mounted is entered in the system logical name table, and the volume is made accessible to all other users provided that UIC-based protection allows access to the volume. To specify MNT\$M_SYSTEM, the caller must have SYSNAM privilege. MNT\$M_SYSTEM applies only to disks.
MNT\$M_WRITECHECK	Write checks are to be performed after all write operations.
MNT\$M_WRITETHRU	Write-back caching is disabled so that file headers are written back to disk with every write operation. If not specified, file headers are cached until the file is closed. Caching file headers improves performance at the risk of losing written data if the system fails. MNT\$M_WRITETHRU applies only to disks.
MNT\$M_NOMNTVER	The volume is not marked as a candidate for automatic mount verification. If not specified, the volume is marked as a candidate for mount verification. MNT\$M_NOMNTVER applies only to disks.
MNT\$M_NOCACHE	All caching associated with the volume is turned off. Specifying MNT\$M_NOCACHE is equivalent to (1) specifying MNT\$M_WRITETHRU, (2) specifying a value of 1 for the item descriptor MNT\$_FILEID, and (3) specifying a value of 0 for the item descriptors MNT\$M_EXTENT and MNT\$M_QUOTA. MNT\$M_NOCACHE applies only to disks.
MNT\$M_NOAUTO	Automatic volume labeling (AVL) and automatic volume recognition (AVR) are to be disabled. If MNT\$M_NOAUTO is specified, the operator must enter commands from the console to process each additional volume in a volume set. When a volume is finished processing, the operator specifies the drive on which the next volume is loaded and the label name of the next volume. The user may want to use MNT\$M_NOAUTO to disable AVL and AVR when not reading a volume set sequentially. The user can enable AVL and AVR by specifying MNT\$M_INIT_CONT. MNT\$M_NOAUTO applies only to magnetic tapes.
MNT\$M_INIT_CONT	Additional volumes in the volume set are to be initialized without operator intervention. \$MOUNT initializes new volumes with the protections specified for the first magnetic tape of the volume set and creates unique volume label names for up to 99 volumes in a volume set.

System Service Descriptions

\$MOUNT

Option	Description
	<p>If MNT\$M_INIT_CONT is specified, the user must allocate multiple magnetic tape drives to the volume set. If \$MOUNT switches to a drive that has no magnetic tape loaded or has the wrong magnetic tape loaded, or if \$MOUNT tries to read a magnetic tape that is not loaded, it notifies the operator to load the correct magnetic tape. \$MOUNT will dismount and unload volumes as soon as they have been read or written. The operator can load the next volume in the volume set before the current reel of the volume set reaches the end of the magnetic tape.</p> <p>If writing to the volume set, \$MOUNT automatically: (1) switches to the next magnetic tape drive; (2) initializes that magnetic tape with the same volume name and protection as specified in the volume labels of the first volume in the set; and (3) notifies the operator that the switch has occurred. If reading the volume set, \$MOUNT generates the label for the next volume in the volume set and reads that volume.</p> <p>The label name that \$MOUNT generates for each additional volume in the volume set consists of 6 characters: the first four characters are the same as the first four characters of the label name of the previous volume; the fifth and sixth characters represent the number of the volume in the volume set.</p>
MNT\$M_CLUSTER	<p>MNT\$M_INIT_CONT applies only to magnetic tapes.</p> <p>The volume is to be mounted for clusterwide access; that is, every node on the cluster can access the volume. \$MOUNT mounts the volume first on the caller's node and then on every other node in the existing VAXcluster.</p> <p>Only system or group volumes can be mounted clusterwide. If MNT\$M_GROUP or MNT\$M_SYSTEM is not specified, \$MOUNT mounts the volume as a system volume, providing the caller has SYSNAM privilege. To mount a group volume clusterwide, the caller must have GRPNAM privilege. To mount a system volume clusterwide, the caller must have SYSNAM privilege.</p> <p>MNT\$M_CLUSTER has no effect if the system is not a member of a VAXcluster. MNT\$M_CLUSTER applies only to disks.</p>

System Service Descriptions

\$MOUNT

Option	Description
MNT\$M_OVR_VOLO	<p>The volume label's owner identifier field is not to be processed. \$MOUNT reads volume owner and protection information from the volume owner field of the volume labels.</p> <p>VAX/VMS requires that the user specify MNT\$M_OVR_VOLO to process magnetic tapes when all of the following conditions exist: (1) the volume was created on a DIGITAL operating system other than VAX/VMS; (2) the volume was initialized with a protection specified; and (3) the volume conforms to the Version 3 ANSI label standard.</p> <p>To specify MNT\$M_OVR_VOLO, the caller must either have VOLPRO privilege or own the volume. MNT\$M_OVR_VOLO applies only to tapes.</p>
MNT\$M_TAPE_DATA_WRITE	<p>Enables the tape controller's write cache for this device. Enabling the write cache improves data throughput for write operations. By default, the tape controller's write cache is disabled for the device.</p> <p>This option applies only to tape systems that support a write cache.</p>

MNT\$_ACCESSED

MNT\$_ACCESSED specifies the number of directories that will be in use, concurrently, on the volume. The buffer must contain a longword integer value in the range 0 to 255. This value overrides the number of directories specified when the volume was initialized. To specify MNT\$_ACCESSED, the caller must have OPER privilege. MNT\$_ACCESSED applies only to disks.

MNT\$_PROCESSOR

For magnetic tapes and Files-11 Structure Level 1 disks, MNT\$_PROCESSOR specifies the name of the ancillary control process (ACP) that is to process the volume. The specified ACP overrides the default ACP that is associated with the device.

For Files-11 Structure Level 2 disks, MNT\$_PROCESSOR controls block cache allocation.

To specify MNT\$_PROCESSOR, the caller must have OPER privilege.

The buffer must contain a character string specifying either the string UNIQUE, a device name, or a file specification. Following is a description of the action taken for each of these cases.

System Service Descriptions

\$MOUNT

String	Description
UNIQUE	For magnetic tapes and Files-11 Structure Level 1 disks, UNIQUE specifies that \$MOUNT create a new process to execute a copy of the default ACP image that is associated with the device specified by the MNT\$_DEVNAM item code. For Files-11 Structure Level 2 disks, UNIQUE allocates a separate block cache.
ddcu:	For magnetic tapes and Files-11 Structure Level 1 disks, ddcu: specifies that \$MOUNT use the ACP process that is currently being used by the device ddcu: . The device specified must be in the format ddcu: , for example, DRA3: . For Files-11 Structure Level 1 disks, ddcu: specifies that \$MOUNT takes the block allocation from the specified device.
file-spec	Specifies that \$MOUNT create a new process to execute the ACP image with the file specification file-spec . Wildcard characters are not allowed in the file specification. The file must be in the disk and directory specified by the logical name SYS\$SYSTEM:. This operation requires CMKRNL privilege.

MNT\$_VOLSET

MNT\$_VOLSET specifies the name of a volume set. The buffer must contain a character string of from 1 to 12 alphanumeric characters, which is the volume set name.

When MNT\$_VOLSET is specified, volumes specified by the MNT\$_VOLNAM item code are bound into a new volume set or added to an existing volume set, depending on whether the name specified by MNT\$_VOLSET is a new or already existing name.

When MNT\$_VOLSET is specified to add volumes to an existing volume set, the root volume (RVN1) must either (1) already be mounted or (2) must be specified first (by the MNT\$_DEVNAM and MNT\$_VOLNAM item codes) in the item list.

When MNT\$_VOLSET is specified to create a new volume set, the first volume specified (by the MNT\$_DEVNAM and MNT\$_VOLNAM item codes) in the item list becomes the root volume.

MNT\$_BLOCKSIZE

MNT\$_BLOCKSIZE specifies the default block size for tape volumes. The buffer must contain a longword integer value in the range 20 to 65532 bytes for VAX RMS operations or 10 to 65534 bytes for operations that do not use VAX RMS. MNT\$_BLOCKSIZE applies only to tapes.

If MNT\$_BLOCKSIZE is not specified, the default block size is 2048 bytes for Files-11 tape volumes and 512 bytes for foreign and unlabeled tapes.

MNT\$_BLOCKSIZE must be specified when mounting (1) tapes that do not have ANSI HDR2 labels, (2) tapes to which data will be written from compatibility mode, and (3) tapes that are to contain records whose size is larger than the default value.

MNT\$_DENSITY

MNT\$_DENSITY specifies the density at which data is to be written to a foreign or unlabeled tape. The buffer must contain a longword value that specifies one of the following legal densities: 800, 1600, or 6250 bpi. MNT\$_DENSITY applies only to tapes.

System Service Descriptions

\$MOUNT

The specified density will be used only if (1) the tape is foreign or unlabeled and (2) the first operation is a write.

MNT\$_EXTENT

MNT\$_EXTENT specifies the size of the extent cache in units of extent pointers. The buffer must contain a longword value, which specifies this size. To specify MNT\$_EXTENT, OPER privilege is required. A value of 0 (the default) disables caching. MNT\$_EXTENT applies only to disks.

MNT\$_FILEID

MNT\$_FILEID specifies the size of the file-id cache in units of file numbers. The buffer must contain a longword value, which specifies this size. To specify MNT\$_FILEID, OPER privilege is required. A value of 1 disables caching. MNT\$_FILEID applies only to disks.

MNT\$_LIMIT

MNT\$_LIMIT specifies the maximum amount of free space in the extent cache. The buffer must contain a longword value, which specifies the amount of free space in units of tenths of a percent of the disk's total free space. MNT\$_LIMIT applies only to disks.

MNT\$_OWNER

MNT\$_OWNER specifies the UIC to be assigned ownership of the volume. The buffer must contain a longword octal value, which is the UIC. If the volume is Files-11 structured, the specified value overrides the ownership recorded on the volume. Either VOLPRO privilege or ownership of the volume is required to assign a UIC to a Files-11 structured volume.

MNT\$_VPROT

MNT\$_VPROT specifies the protection to be assigned to the volume. The buffer must contain a longword protection mask, which specifies the four types of access allowed to the four categories of user.

The protection mask consists of four 4-bit fields. Each field grants or denies read, write, logical, and physical access to a category of users. Cleared bits grant access; set bits deny access. The following diagram depicts the structure of the protection mask:

WORLD				GROUP				OWNER				SYSTEM			
P	L	W	R	P	L	W	R	P	L	W	R	P	L	W	R
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

ZK-1715-84

If MNT\$_VPROT is not specified or is specified as 0, the volume receives the protection that it was assigned when it was initialized. To specify MNT\$_VPROT for a Files-11 structured volume, the caller must either own the volume or have VOLPRO privilege.

MNT\$_QUOTA

MNT\$_QUOTA specifies the size of the quota record cache in units of quota records. The buffer must contain a longword value, which is this size. To specify MNT\$_QUOTA, OPER privilege is required. A value of 0 disables caching. MNT\$_QUOTA applies only to disks.

System Service Descriptions

\$MOUNT

MNT\$_RECORDSIZ

MNT\$_RECORDSIZ specifies the number of characters in each record. MNT\$_RECORDSIZ is used with MNT\$_BLOCKSIZE to specify the data formats for foreign volumes. The buffer must contain a longword value that is less than or equal to the block size. MNT\$_RECORDSIZ applies only to tapes.

If MNT\$_RECORDSIZ is not specified, the record size is assumed to be equal to the block size.

MNT\$_SHAMEM

This item code applies only to the volume shadowing option. See the *VAX/VMS Volume Shadowing Manual*.

MNT\$_SHAMEM_COPY

This item code applies only to the volume shadowing option. See the *VAX/VMS Volume Shadowing Manual*.

MNT\$_SHAMEM_MGCOPY

This item code applies only to the volume shadowing option. See the *VAX/VMS Volume Shadowing Manual*.

MNT\$_SHANAM

This item code applies only to the volume shadowing option. See the *VAX/VMS Volume Shadowing Manual*.

MNT\$_WINDOW

MNT\$_WINDOW specifies the number of mapping pointers to be allocated for file windows. The buffer must contain a longword value in the range 7 to 80. This value overrides the default value that was applied when the volume was initialized. MNT\$_WINDOW applies only to disks.

When a file is opened, the file system uses the mapping pointers to access the data in the file. To specify MNT\$_WINDOW, OPER privilege is required.

MNT\$_EXTENSION

MNT\$_EXTENSION specifies the number of blocks by which files will be extended. The buffer must contain a longword value in the range 0 to 65,535. MNT\$_EXTENSION applies only to disks.

MNT\$_COMMENT

MNT\$_COMMENT specifies text to be associated with an operator request. The buffer must contain a character string of no more than 78 characters. This text will be printed on the operator's console if an operator request is issued for the device being mounted.

DESCRIPTION

To mount a particular volume, the caller must either own or have privilege to access the specified volume or volumes. The privileges required depend on the operation and are listed with the item codes that specify the operation.

The calling process must have TMPMBX or PRMMBX privilege to perform an operator-assisted mount.

\$MOUNT uses the following system resources to mount volumes with group or systemwide access allowed:

- Nonpaged pool
- System dynamic pool

CONDITION VALUES RETURNED

SS\$_NORMAL	Service successfully completed.
SS\$_ACCVIO	The item list or an address specified in the item list cannot be accessed.
SS\$_BADPARAM	A buffer length of zero was specified with a nonzero item code; an illegal item code was specified; or no device was specified.
SS\$_NOGRPNAM	The caller does not have GRPNAM privilege.
SS\$_NOSYSNAM	The caller does not have SYSNAM privilege.
SS\$_NOOPER	The caller does not have the required OPER privilege.
SS\$_NOPRIV	The caller does not have sufficient privilege to access a specified volume.
SS\$_NOSUCHDEV	Warning. The specified device does not exist on the host system.

\$MOUNT may also return a condition value that is specific to the mount facility. These condition values are defined by the symbolic definition macro \$MOUNDEF. For information on how to obtain these symbolic codes see Section 2.3.

The Magnetic Tape Accessibility service allows installations to provide their own routine to interpret and output the accessibility field in the VOL1 and HDR1 labels of an ANSI labeled magnetic tape. (ANSI refers to the *American National Standard for Magnetic Tape Labels and File Structure for Information Interchange—ANSI, x3.27-1978*.)

Longword condition value. All system services return (by immediate value) a condition value in R0. For the input of a label, condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED". For the output of a label, the value returned in the low byte in R0 is the **access_char** to write to the label.

VMS Usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**

System Service Descriptions

\$MTACCESS

mechanism: **by value**

Decimal equivalent of the ANSI standard version read from the VOL1 label. The **std_version** argument is a longword containing the standard version number.

access_char

VMS Usage: **longword_unsigned**

type: **longword (unsigned)**

access: **read only**

mechanism: **by value**

Accessibility character specified by the user. The **access_char** argument is a byte containing the accessibility character used for the output of labels.

access_spec

VMS Usage: **longword_unsigned**

type: **longword (unsigned)**

access: **read only**

mechanism: **by value**

Value specifying whether the accessibility character passed in **access_char** was specified by the user. The **access_spec** argument is a byte containing one of the following values:

Value	Meaning
MTASK_CHARVALID	Yes
MTASK_NOCHAR	No

This argument is used only for the output of labels.

type

VMS Usage: **longword_unsigned**

type: **longword (unsigned)**

access: **read only**

mechanism: **by value**

Type of accessibility field to process. The **type** argument is a byte containing one of the following values:

Value	Meaning
MTASK_INVOL1	Input a VOL1 label
MTASK_INHDR1	Input a HDR1 label
MTASK_OUTVOL1	Output a VOL1 label
MTASK_OUTHDR1	Output a HDR1 label

DESCRIPTION

The \$MTACCESS service allows installations to provide their own routine to interpret and output the accessibility field in the VOL1 and HDR1 labels of ANSI labeled magnetic tapes. The installation can override the default routine by setting the flag LOADMTACCESS with SYSGEN. When the system reboots, the routine MTACCESS.EXE is loaded into the system and the system service dispatch vectors are set to call the installation's routine instead of the default routine.

System Service Descriptions

\$MTACCESS

The default installation routine first checks the ANSI standard version of the label. For magnetic tapes with a version number of 3 or less, the routine outputs either a blank or the character specified by the user. On input of these magnetic tapes, the routine checks for a blank and returns the value `SS$_FILACCERR` if the field is not blank.

For magnetic tapes with a version number greater than 3, the routine outputs either the character specified by `access_char` or an ASCII 1 if no character was specified. On input of these magnetic tapes, the routine checks for a blank. If the field is blank, `R0` is set to 0. In that case, the user is given full access and VAX/VMS protection is not checked. If the field contains an ASCII 1 and the `VOL1 IMPLEMENTATION IDENTIFIER` field contains the VAX/VMS system code, `R0` is set to `SS$_NORMAL`. In that case, the VAX/VMS protection is checked.

If the field is not blank and does not contain an ASCII 1, `R0` is set to `SS$_FILACCERR`, which forces the user to override the accessibility checking, and allows the magnetic tape file system to check VAX/VMS protection.

The following summarizes the results of label input check:

Contents of R0	Result
<code>SS\$_NORMAL</code>	Check the VAX/VMS protection on the magnetic tape
0	Give the user full access. VAX/VMS protection is not checked
<code>SS\$_FILACCERR</code>	Check for explicit override, then check VAX/VMS protection

Please note that the default accessibility routine does not output `SS$_NOVOLACC` or `SS$_NOFILACC`. These statuses are included for the installations use, and the magnetic file system knows how to deal with them.

The magnetic tape file system has been enhanced to call `$MTACCESS` to process the accessibility field in the `VOL1` and `HDR1` labels. After a call to the system service, the magnetic tape file system checks to ensure that the installation did not move the magnetic tape. If the magnetic tape was moved, the magnetic tape file system completes the current operation with a `SS$_TAPEPOSLOST` error. Finally, it processes the remainder of the label according to the status returned by `$MTACCESS`.

Because accessibility is an installation provided routine, VAX/VMS cannot determine which users have the authority to override the processing of this field. However, the magnetic tape file system allows only operator class users to deal with blank magnetic tapes so that a user must have both `OPER` and `VOLPRO` privileges to initialize or mount blank magnetic tapes.

CONDITION VALUES RETURNED

<code>SS\$_NORMAL</code>	Successful completion.
<code>SS\$_FILACCERR</code>	Check for explicit override.
<code>SS\$_NOFILACC</code>	The user has no access to the file.
<code>SS\$_NOVOLACC</code>	The user has no access to the volume.

\$NUMTIM—Convert Binary Time to Numeric Time

The Convert Binary Time to Numeric Time service converts an absolute or delta time from 64-bit system time format to binary integer date and time values.

FORMAT **SY\$NUMTIM** *timbuf*,[*timadr*]

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS *timbuf*

VMS Usage: **vector_word_unsigned**
type: **word (unsigned)**
access: **write only**
mechanism: **by reference**

Buffer into which \$NUMTIM writes the converted date and time. The **numtim** argument is the address of a 7-word structure. The following diagram depicts the fields in this structure:

31	15	0
month of year	year since 0	
hour of day	day of month	
second of minute	minute of hour	
	hundredths of second	

ZK-1716-84

If the **timadr** argument specifies a delta time, \$NUMTIM returns 0 in the **year since 0** and **month of year** fields. It returns in the **day of month** field the number of days specified by the delta time, which must be less than 10,000 days.

System Service Descriptions

\$NUMTIM

timadr

VMS Usage: **date_time**
type: **quadword (unsigned)**
access: **read only**
mechanism: **by reference**

The 64-bit time value to be converted. The **timadr** argument is the address of a quadword containing this time. A positive time value represents an absolute time, while a negative time value indicates a delta time.

If **timadr** is not specified, \$NUMTIM returns the current system time.

If **timadr** specifies 0, \$NUMTIM returns the base date (November 17, 1858).

CONDITION VALUES RETURNED

SS\$_NORMAL

Service successfully completed.

SS\$_ACCVIO

The 64-bit time value cannot be read by the caller, or the buffer cannot be written by the caller.

SS\$_IVTIME

The specified delta time is equal to or greater than 10,000 days.

\$PARSE_ACL—Parse Access Control List Entry

The Parse Access Control List Entry service parses the specified text string and converts it into the binary representation for an access control list entry (ACE).

FORMAT **SYS\$PARSE_ACL** *aclstr*, *aclent*, [*errpos*], [*accnam*]

RETURNS

VMS Usage: **cond_value**
 type: **longword (unsigned)**
 access: **write only**
 mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED".

ARGUMENTS

aclstr

VMS Usage: **char_string**
 type: **character-coded text string**
 access: **read only**
 mechanism: **by descriptor—fixed length string descriptor**

Formatted ACE that is parsed when \$PARSE_ACL completes execution. The *aclstr* argument is the address of a string descriptor pointing to the text string to be parsed.

aclent

VMS Usage: **char_string**
 type: **character-coded text string**
 access: **write only**
 mechanism: **by descriptor—fixed length string descriptor**

Description of the ACE that is parsed when \$PARSE_ACL completes execution. The *aclent* argument is the address of a descriptor pointing to the buffer in which the ACE is written. The first byte of the buffer contains the length of the ACE; the second byte contains a value that identifies the type of ACE, which in turn defines the format of the ACE. See the SYS\$FORMAT_ACL service for information on the ACE types and their associated formats.

errpos

VMS Usage: **word_unsigned**
 type: **word (unsigned)**
 access: **write only**
 mechanism: **by reference**

Number of characters from *aclstr* processed by SYS\$PARSE_ACL. The *errpos* argument is the address of a word that receives the number of characters actually processed by the service. If the service fails, this count points to the failing point in the string.

System Service Descriptions

\$PARSE_ACL

accnam

VMS Usage: **access_bit_names**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Names of the bits in the access mask when executing \$PARSE_ACL. The **accnam** argument is the address of an array of 32 quadword descriptors that define the names of the bits in the access mask. Each element points to the name of a bit. The first element names bit 0, the second element names bit 1, and so on. If **accnam** is omitted, the following names are used:

Bit 0	READ
Bit 1	WRITE
Bit 2	EXECUTE
Bit 3	DELETE
Bit 4	CONTROL
Bit 5	BIT_5
Bit 6	BIT_6
.	.
.	.
Bit 31	BIT_31

DESCRIPTION	The Parse Access Control List Entry service parses the specified text string and converts it into the binary representation for an access control list entry.
--------------------	---

CONDITION VALUES RETURNED

SS\$_NORMAL
SS\$_ACCVIO
SS\$_IVACL

Service successfully completed.

The string or its descriptor cannot be read by the caller, or the buffer descriptor cannot be read by the caller, or the buffer cannot be written by the caller, or the buffer is too small to hold the ACL entry.

The format of the access control list entry is not valid.

\$PURGWS—Purge Working Set

The Purge Working Set service removes a specified range of pages from the calling process's current working set to make room for pages required by a new program segment. However, the Adjust Working Set Limit (\$ADJWSL) service is the preferred mechanism for controlling a process's use of physical memory resources.

FORMAT	SYSPURGWS <i>inadr</i>
---------------	-------------------------------

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENT	<i>inadr</i>
-----------------	---------------------

VMS Usage: **address_range**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Starting and ending virtual addresses of the range of pages to be purged. The ***inadr*** is the address of a two-longword array containing, in order, the starting and the ending process virtual addresses. Only the virtual page number portion of each virtual address is used; the low-order 9 bits are ignored.

The \$PURGWS service locates pages within the specified range and removes them if they are in the working set.

If the starting and ending virtual addresses are the same, only that single page is purged.

To purge the entire working set, specify a range of pages from 0 through 7FFFFFFF; in this case, the image will continue to execute and pages will be faulted back into the working set as they are needed.

CONDITION VALUES RETURNED

SS\$_NORMAL
SS\$_ACCVIO

Service successfully completed.

The input address array cannot be read by the caller.

System Service Descriptions

\$PUTMSG

\$PUTMSG—Put Message

The Put Message service writes one or more error messages to SYS\$ERROR (and to SYS\$OUTPUT if it is different than SYS\$ERROR). \$PUTMSG is a generalized message formatting and output routine used by VAX/VMS to write informational and error messages to processes.

FORMAT **SYS\$PUTMSG** *msgvec* ,[*actrtn*] ,[*facnam*] ,[*actprm*]

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

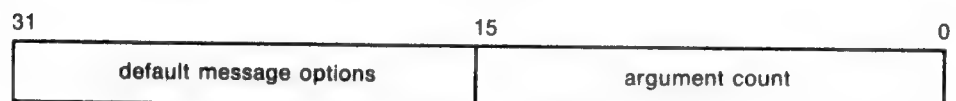
Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS *msgvec*

VMS Usage: **cntrlblk**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Message argument vector specifying the message(s) to be written and options that \$PUTMSG is to use in writing the message(s). The **msgvec** argument is the address of the message vector.

The message vector consists of one longword followed by one or more message descriptors, one descriptor per message. The following diagram depicts the contents of the first longword.



ZK-1717-84

Message Vector Fields

argument count

This word-length field specifies the total number of longwords in the message vector, not including the first longword (of which it is a part).

default message options

This word-length field specifies which message component(s) are to be written. The **default message options** field is a word-length bit vector wherein a bit, when set, specifies that the corresponding message component is to be written. Refer to the Description section for a description of each of these components.

System Service Descriptions

\$PUTMSG

The following list gives the significant bit numbers. Note that the bit numbers shown (0, 1, 2, 3) are the bit positions from the beginning of the word; however, since the word is the second word in the longword, the number 16 should be added to each bit number to specify its exact offset within the longword.

Bit	Value	Description
0	1	Include message text
	0	Do not include message text
1	1	Include mnemonic name for message text
	0	Do not include mnemonic name for message text
2	1	Include severity level indicator
	0	Do not include severity level indicator
3	1	Include facility prefix
	0	Do not include facility prefix

Bits 4 through 15 must be 0.

The default setting specified by the **default message options** field can be overridden for any or all messages by specifying different options in the **new message options** field of any subsequent message descriptor. When **new message options** is specified, the options it specifies become the new default settings for all remaining messages until **new message options** is specified again.

\$PUTMSG passes the **default message flags** field to the \$GETMSG service as the **flags** argument.

If the **default message flags** field is not specified, the default message options for the process are used; the process default message options can be set using the DCL command SET MESSAGE.

The Description section shows the format that \$PUTMSG uses to write these message components.

Following the first longword of the message vector are one or more message descriptors. A message descriptor may have one of four possible formats, depending on the type of message it describes. There are four types of messages:

- 1 User-supplied messages
- 2 System messages
- 3 VAX RMS messages
- 4 System exception messages

System Service Descriptions

\$PUTMSG

Message Descriptor for User-Supplied Messages

31	15	0
message code		
new message options		FAO parameter count
First FAO parameter		
Second FAO parameter		
.		
.		
.		

ZK-1718-84

Fields in Message Descriptor for User-Supplied Messages

message code

Longword value that uniquely identifies the message. The Description section discusses the message code, and the description of the Message Utility in the *VAX/VMS Utilities Reference Volume* explains how to create message codes.

FAO parameter count

Word-length value specifying the number of longword FAO parameters that follow in the message descriptor. The number of FAO parameters needed depends on the FAO directives used in the message text; some FAO directives require one or more parameters, while some directives require none.

new message options

Word-length bit vector specifying new message options for the current message. The contents and format of this field are identical to that of the default message options field.

FAO parameter

Longword value that is used by an FAO directive appearing in the message text. The FAO parameters listed in the message descriptor must appear in the order in which they will be used by the FAO directives in the message text.

Message Descriptor for System Messages

31	0
message code	

ZK-1719-84

Fields in Message Descriptor for System Messages

message code

Longword value that uniquely identifies the message. The facility number field in the message code identifies the facility associated with the message. A system message has a facility number of 0. The **FAO parameter count**, **new message options**, and **FAO parameter** fields cannot be specified. Each

System Service Descriptions

\$PUTMSG

longword following the **message identification** field in the message vector will be interpreted as another message identification.

Message Descriptor for VAX RMS Messages

31

0

message code
RMS Status Value (STV)

ZK-1720-84

Fields in Message Descriptor for VAX RMS Messages

message code

Longword value that uniquely identifies the message. The facility number field in the message code identifies the facility associated with the message. An RMS message has a facility number of 1. The **FAO parameter count**, **new message options**, and **FAO parameter** fields cannot be specified. The longword following the **message identification** field in the message vector will be interpreted as an STV.

RMS Status Value

Longword containing an STV for use by an RMS message that has an associated STV value. \$PUTMSG uses the STV value as an FAO parameter or as another message identification, depending on the RMS message identified by the **message identification** field. If the RMS message does not have an associated STV, \$PUTMSG ignores the STV longword in the message descriptor.

Message Descriptor for System Exception Messages

31

0

message code
First FAO parameter
Second FAO parameter
.
.

ZK-1721-84

Fields in Message Descriptor for System Exception Messages

message code

Longword value that uniquely identifies the message. The facility number field in the message code identifies the facility associated with the message. A system exception message has a facility number of 0. The **FAO parameter count** and **new message options** fields cannot be specified. The longword(s) following the **message code** field in the message vector will be interpreted as FAO parameters.

System Service Descriptions

\$PUTMSG

actrtn

VMS Usage: **procedure**
type: **procedure entry mask**
access: **call without stack unwinding**
mechanism: **by reference**

User-supplied action routine to be executed during message processing. The **actrtn** argument is the address of the entry mask of this routine.

The action routine receives control after a message is formatted but before it is actually written to the user.

If **actrtn** is not specified or is specified as 0 (the default), no action routine executes.

Since \$PUTMSG writes messages only to SYS\$ERROR and SYS\$OUTPUT, an action routine is useful, for example, when output must be directed to a file.

facnam

VMS Usage: **char_string**
type: **character-coded text string**
access: **read only**
mechanism: **by descriptor—fixed length string descriptor**

Facility prefix to be used in the first or only message written by \$PUTMSG. The **facnam** argument is the address of a character string descriptor pointing to this facility prefix.

If **facnam** is not specified, \$PUTMSG uses the default facility prefix that is associated with the message.

actprm

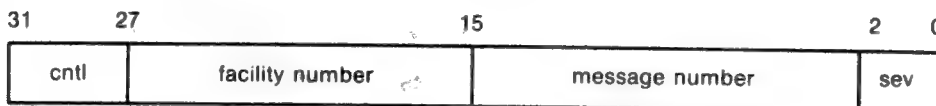
VMS Usage: **user_arg**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Parameter to be passed to the action routine. The **actprm** is a longword value containing this parameter. If **actprm** is not specified, no parameter is passed.

Note that the first argument that is passed to the action routine is the address of a character string descriptor pointing to the message text; the parameter specified by **actprm** is the second.

DESCRIPTION

In VAX/VMS, a message is identified by a longword value, which is called the **message code**. To construct a message code, values for its four fields are specified using the Message Utility. The following diagram depicts the longword message code.



ZK-1722-84

Thus, each message has a unique longword value associated with it: its message code. A symbolic name may be given to this longword value using the Message Utility. Such a symbolic name is called the **message symbol**.

System Service Descriptions

\$PUTMSG

The Message Utility describes how to construct a message symbol according to the conventions for VAX/VMS messages. Basically, the message symbol has two parts: (1) a facility prefix, which is an abbreviation of the name of the facility with which the message is associated and (2) a mnemonic name for the message text, which serves to hint at the nature of the message. These two parts are separated by an underscore character (_) in the case of a user-constructed message and by a dollar sign/underscore (\$_) in the case of system messages.

The message components that are written by \$PUTMSG are derived both from the message code and from the message symbol. Refer to the description of the Message Utility in the *VAX/VMS Utilities Reference Volume* for additional information about both the message code and the message symbol.

The \$PUTMSG service writes the message components in the following format:

`%FACILITY-L-IDENT, message text`

The following list describes the information provided by each segment of the message that the \$PUTMSG service writes.

%	Is the prefix used for the first message written. The hyphen (-) is the prefix used for the remaining messages.
FACILITY	Is the facility prefix taken from the message symbol. This facility prefix may be overridden by a facility prefix specified in the <code>facnam</code> argument in the call to \$PUTMSG.
L	Is the severity level indicator. The severity level indicator is taken from the message code.
IDENT	Is a mnemonic name for the message text. The ident is taken from the message symbol.
message text	Is the message text. The message text is specified in the message source file.

The \$PUTMSG service does not check the length of the argument list and therefore cannot return the `SS$_INSFARG` (insufficient arguments) condition value. Be sure you specify the required number of arguments.

If an error occurs while \$PUTMSG calls the Formatted ASCII Output (\$FAO) service, FAO parameters specified in the message vector will not appear in the output.

The \$PUTMSG service cannot be called from kernel mode.

CONDITION VALUES RETURNED

`SS$_NORMAL`

Service successfully completed.

System Service Descriptions

\$PUTMSG

EXAMPLES

```
1 VECTOR: .LONG 3           ; argument count & null msg. flags
          .LONG SS$_ABORT   ; abort message
          .LONG RMS$_FNF    ; file not found message
          .LONG 0           ; null stv parameter
          .
          $PUTMSG_S -
            MSGVEC=VECTOR
```

The above example shows a segment of a program used to request \$PUTMSG to write the following messages to the current SYS\$OUTPUT device (and to SYS\$ERROR, if it is different):

- The complete message associated with the system status code SS\$_ABORT (%SYSTEM-F-ABORT, abort)
- The complete message associated with the system status code RMS\$_FNF (-RMS-E-FNF, file not found)

```
2 INTEGER STATUS,
  2 OLDHND
CHARACTER*5 NUM
INCLUDE '($SDEF)'
INCLUDE '($LIBDEF)'
INTEGER LIB$GET_INPUT,
  2 LIB$ESTABLISH,
  2 SYS$GETJPI
EXTERNAL ERR
OPEN (UNIT = 1,
  2 TYPE = 'NEW',
  2 CARRIAGECONTROL = 'LIST',
  2 FILE = 'ERROR.LOG')
OLDHND = LIB$ESTABLISH (ERR)
! this routine executes successfully
STATUS = LIB$GET_INPUT (NUM, 'NUM: ')
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
! this routine fails with insufficient arguments
STATUS = SYS$GETJPI(,)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
END

INTEGER FUNCTION ERR (SIGARGS,
  2 MECHARGS)

INTEGER SIGARGS(*),
  2 MECHARGS(*)
INTEGER NEWSIGARGS(10), ! must specify a length for
                        ! array so choose one large enough
                        ! to cover any eventuality

  2 ELEMENT
INCLUDE '($SDEF)'
EXTERNAL PUT_LINE
INTEGER PUT_LINE

! Get rid of last two elements in SIGARGS (the PC and PSL),
! then pad NEWSIGARGS with zeros.
ELEMENT = 1
NEWSIGARGS(ELEMENT) = 10
DO I = 1, SIGARGS(1) - 2
  ELEMENT = ELEMENT + 1
  NEWSIGARGS (ELEMENT) = SIGARGS (ELEMENT)
END DO
DO I = ELEMENT + 1, 10
```

System Service Descriptions

\$PUTMSG

```
      ELEMENT = ELEMENT + 1
      NEWSIGARGS (ELEMENT) = 0
    END DO
    CALL SYS$PUTMSG (NEWSIGARGS, PUT_LINE,)

    ERR = SS$_RESIGNAL
      ! could use CONTINUE and let $PUTMSG
      ! write the message

    END

    INTEGER FUNCTION PUT_LINE (LINE)
    CHARACTER*(*) LINE
    PUT_LINE = 0      ! since you're resignalling, don't let
      ! SYS$PUTMSG write the error.

    WRITE (UNIT = 1,
     2     FMT = '(A)') LINE
    END
```

The VAX FORTRAN example above uses \$PUTMSG to write any error messages to a file (ERROR.LOG) as well as to the terminal.

\$Q10

For additional information about system service completion, refer to the Synchronize (\$SYNCH) service and to Section 2.5.

SYS\$QIO *[efn],chan,func[,iosb][,astadr][,astprm]
[,p1][,p2][,p3][,p4][,p5][,p6]*

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED"

VMS Usage: channel
type: word (unsigned)
access: read only

DESCRIPTION

\$QIO operates only on assigned I/O channels and only from access modes that are equal to or more privileged than the access mode from which the original channel assignment was made.

\$QIO uses the following system resources:

- The process's quota for buffered I/O limit (BIOLM) or direct I/O limit (DIOLM)
- The process's buffered I/O byte count (BYTLM) quota
- The process's AST limit (ASTLM) quota, if an AST service routine is specified
- System dynamic memory is required to construct a data base to queue the I/O request
- Additional memory may be required on a device-dependent basis

For \$QIO, completion can be synchronized by (1) specifying the **astadr** argument to have an AST routine execute when the I/O completes or (2) by calling the Synchronize (\$SYNCH) service to await completion of the I/O operation. \$QIOW completes synchronously, and it is the best choice when synchronous completion is required.

For information on using \$QIO for network operations, refer to the *VAX/VMS Networking Manual*.

CONDITION VALUES RETURNED

SS\$_NORMAL	Service successfully completed. The I/O request was successfully queued.
SS\$_ABORT	A network logical link was broken.
SS\$_ACCVIO	Either the I/O status block cannot be written by the caller, or the parameters for device-dependent function codes are incorrectly specified.
SS\$_DEVOFFLINE	The specified device is offline and not currently available for use.
SS\$_EXQUOTA	The process has (1) exceeded its AST limit (ASTLM) quota, (2) exceeded its buffered I/O byte count (BYTLM) quota, (3) exceeded its buffered I/O limit (BIOLM) quota, (4) exceeded its direct I/O limit (DIOLM) quota, or (5) requested a buffered I/O transfer smaller than the buffered byte count quota limit (BYTLM), but when added to other current buffer requests, the buffered I/O byte count quota was exceeded.
SS\$_ILLEFC	An illegal event flag number was specified.
SS\$_INSFMEM	Insufficient system dynamic memory is available to complete the service.
SS\$_IVCHAN	An invalid channel number was specified, that is, a channel number of 0 or a number larger than the number of channels available.

System Service Descriptions

\$QIO

SS\$_NOPRIV	The specified channel does not exist, was assigned from a more privileged access mode, or the process does not have the necessary privileges to perform the specified functions on the device associated with the specified channel.
SS\$_UNASEFC	The process is not associated with the cluster containing the specified event flag.
SS\$_LINKABORT	The network partner task aborted the logical link.
SS\$_LINKDISCON	The network partner task disconnected the logical link.
SS\$_PATHLOST	The path to the network partner task node was lost.
SS\$_PROTOCOL	A network protocol error occurred. This is most likely due to a network software error.
SS\$_CONNECFAIL	The connection to a network object timed out or failed.
SS\$_FILALRACC	A logical link is already accessed on the channel (that is, a previous connect on the channel).
SS\$_INVLOGIN	The access control information was found to be invalid at the remote node.
SS\$_IVDEVNAM	The NCB has an invalid format or content.
SS\$_LINKEXIT	The network partner task was started, but exited before confirming the logical link (that is, \$ASSIGN to SY\$NET).
SS\$_NOLINKS	No logical links are available. The maximum number of logical links as set for the executor MAXIMUM LINKS parameter was exceeded.
SS\$_NOSUCHNODE	The specified node is unknown.
SS\$_NOSUCHOBJ	The network object number is unknown at the remote node; or for a TASK = connect, the named DCL command procedure file cannot be found at the remote node.
SS\$_NOSUCHUSER	The remote node could not recognize the login information supplied with the connection request.
SS\$_PROTOCOL	A network protocol error occurred. This error is most likely due to a network software error.
SS\$_REJECT	The network object rejected the connection.
SS\$_REMRSRC	The link could not be established because system resources at the remote node were insufficient.
SS\$_SHUT	The local or remote node is no longer accepting connections.
SS\$_THIRDPARTY	The logical link was terminated by a third party (for example, the System Manager).
SS\$_TOOMUCHDATA	The task specified too much optional or interrupt data.
SS\$_UNREACHABLE	The remote node is currently unreachable.

**CONDITION
VALUES
RETURNED
IN THE I/O
STATUS
BLOCK**

Device-specific condition values; the *VAX/VMS I/O Reference Volume* lists these condition values, for each device, in the section describing that device.

System Service Descriptions

\$QIOW

\$QIOW—Queue I/O Request and Wait

The Queue I/O Request and Wait service queues an I/O request to a channel associated with a device.

The \$QIOW service completes synchronously; that is, it returns to the caller after the I/O operation has actually completed.

For asynchronous completion, use the Queue I/O Request (\$QIO) service; \$QIO returns to the caller after queuing the I/O request, without waiting for the I/O operation to be completed.

In all other respects, \$QIOW is identical to \$QIO. Refer to the documentation of \$QIO for all other information about the \$QIOW service.

For additional information about system service completion, refer to the Synchronize (\$SYNCH) service and to Section 2.5.

FORMAT

SYSS\$QIOW *[efn],chan,func[,iosb][,astadr]
[,astprm][,p1][,p2][,p3][,p4][,p5][,p6]*

\$READEF—Read Event Flags

The Read Event Flags service returns the current status of all 32 event flags in a local or common event flag cluster. In addition, the condition value returned indicates whether the specified event flag is set or clear.

FORMAT **SYS\$READEF** *efn, state*

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED".

ARGUMENTS *efn*

VMS Usage: **ef_number**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Number of any event flag in the cluster whose status is to be returned. The **efn** argument is a longword containing this number. Specifying an event flag within a cluster requests that \$READEF return the status of all event flags in that cluster.

There are two local event flag clusters, which are local to the process: cluster 0 and cluster 1. Cluster 0 contains event flag numbers 0 to 31, and cluster 1 contains event flag numbers 32 to 63.

There are two common event flag clusters: cluster 2 and cluster 3. Cluster 2 contains event flag numbers 64 to 95, and cluster 3 contains event flag numbers 96 to 127.

state

VMS Usage: **mask_longword**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

State of all event flags in the specified cluster. The **state** argument is the address of a longword into which \$READEF writes the state (set or clear) of the 32 event flags in the cluster.

System Service Descriptions

\$READEF

CONDITION VALUES RETURNED

SS\$_WASCLR

Service successfully completed. The specified event flag is clear.

SS\$_WASSET

Service successfully completed. The specified event flag is set.

SS\$_ACCVIO

The longword that is to receive the current state of all event flags in the cluster cannot be written by the caller.

SS\$_ILLEFC

An illegal event flag number was specified.

SS\$_UNASEFC

The process is not associated with the cluster containing the specified event flag.

\$REM_HOLDER—Remove Holder Record from Rights Database

Deletes the specified holder record from the target identifier's list of holders.

FORMAT **SY\$REM_HOLDER** *id,holder*

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS *id*

VMS Usage: **rights_id**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Binary value of target identifier whose holder is deleted when \$REM_HOLDER completes execution. The *id* argument is a longword containing the identifier value.

holder

VMS Usage: **rights_holder**
type: **quadword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of holder being deleted when \$REM_HOLDER completes execution. The *holder* argument is the address of a quadword containing the UIC identifier of the holder in the first longword and the value of zero in the second longword.

DESCRIPTION The Remove Holder Record from Rights Database service removes the specified holder record from the target identifier's list of holders.

Write access to the rights database is required to use this service. If the database is in SYS\$SYSTEM (which is the default) SYSPRV privilege is needed to grant write access to the database.

System Service Descriptions

SREM_HOLDER

CONDITION VALUES RETURNED

SS\$_NORMAL

Successful completion.

SS\$_ACCVIO

The **id** or **holder** arguments cannot be read by the caller.

SS\$_INSFMEM

Insufficient process dynamic memory is available to open the rights database.

SS\$_IVIDENT

The specified identifier or holder identifier is of invalid format.

SS\$_NOSUCHID

The specified identifier does not exist in the rights database, or the specified holder identifier does not exist in the rights database.

RMS\$_PRV

The user does not have write access to the rights database.

Since the rights database is an indexed file accessed with VAX RMS, this service may also return RMS status codes associated with operations on indexed files. Please refer to the *VAX Record Management Services Reference Manual* for descriptions of these status codes.

\$REM_IDENT—Remove Identifier from Rights Database

The Remove Identifier from Rights Database service removes the specified identifier record and all its holder records (if any) from the rights database.

FORMAT **SYS\$REM_IDENT** *id*

RETURNS

VMS Usage: **cond_value**
 type: **longword (unsigned)**
 access: **write only**
 mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS

id
 VMS Usage: **rights_id**
 type: **longword (unsigned)**
 access: **read only**
 mechanism: **by value**

Binary value of identifier deleted from rights database when \$REM_IDENT completes execution. The *id* argument is a longword containing the identifier value.

DESCRIPTION

The \$REM_IDENT system service deletes the specified identifier from the rights database. All holder records associated with the identifier are also deleted. In addition, any records in identifiers that the deleted identifier held are also deleted.

Write access to the rights database is required to use this service. If the database is in SYS\$SYSTEM (which is the default) SYSPRV privilege is needed to grant write access to the database.

CONDITION VALUES RETURNED

SS\$_NORMAL	Service is successfully completed.
SS\$_INSFMEM	Insufficient process dynamic memory is available to open the rights database.
SS\$_IVIDENT	The specified identifier is of invalid format.

System Service Descriptions

\$REM_IDENT

SS\$_NOSUCHID

The specified identifier does not exist in the right database.

RMS\$_PRV

The user does not have write access to the rights database.

Since the rights database is an indexed file accessed with VAX RMS, this service may also return RMS status codes associated with operations on indexed files. Please refer to the *VAX Record Management Services Reference Manual* for descriptions of these status codes.

\$RESUME—Resume Process

The Resume Process service (1) causes a process previously suspended by the Suspend Process (\$SUSPND) service to resume execution or (2) cancels the effect of a subsequent suspend request.

FORMAT **SY\$RESUME** [*pidadr*] , [*prcnam*]

RETURNS VMS Usage: **cond_value**
 type: **longword (unsigned)**
 access: **write only**
 mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS ***pidadr***

VMS Usage: **process_id**
 type: **longword (unsigned)**
 access: **modify**
 mechanism: **by reference**

Process identification (PID) of the process that is to be resumed. The ***pidadr*** argument is the address of a longword containing the PID.

The ***pidadr*** argument must be specified to delete processes in other UIC groups.

prcnam

VMS Usage: **process_name**
 type: **character-coded text string**
 access: **read only**
 mechanism: **by descriptor—fixed length string descriptor**

Name of the process to be resumed. The ***prcnam*** is the address of a character string descriptor pointing to the process name, which is a character string of from 1 to 15 characters.

The ***prcnam*** argument can be used to resume only processes in the same UIC group as the calling process. The reason for this is that process names are unique to UIC groups, and VAX/VMS uses the UIC group number of the calling process to interpret the process name specified by the ***prcnam*** argument. The ***pidadr*** argument must be used to delete processes in other UIC groups.

If neither the ***pidadr*** nor ***prcnam*** arguments are specified, the resume request is issued on behalf of the calling process.

System Service Descriptions

\$RESUME

DESCRIPTION Depending on the operation, use of \$RESUME may require the calling process to have certain privilege:

- GROUP privilege is required to resume execution of a process in the same group unless the process has the same UIC as the calling process.
- WORLD privilege is required to resume execution of any process in the system

If one or more resume requests are issued for a process that is not suspended, a subsequent suspend request completes immediately, that is, the process is not suspended. No count is maintained of outstanding resume requests.

**CONDITION
VALUES
RETURNED**

SS\$_NORMAL

Service successfully completed.

SS\$_ACCVIO

The process name string or string descriptor cannot be read by the caller, or the process identification cannot be written by the caller.

SS\$_IVLOGNAM

The specified process name has a length of 0 or has more than 15 characters.

SS\$_NONEXPR

Warning. The specified process does not exist, or an invalid process identification was specified.

SS\$_NOPRIV

The process does not have the privilege to resume the execution of the specified process.

\$REVOKID—Revoke Identifier from Process

The Revoke Identifier from Process service removes the specified identifier from the process's rights list, or from the system rights list. If the identifier is listed as a holder of any other identifier, the appropriate holder records are also deleted.

FORMAT **SYS\$REVOKID** [*pidadr*] , [*prcnam*] , [*id*] , [*name*] , [*prvatr*]

RETURNS

VMS Usage: **cond_value**
 type: **longword (unsigned)**
 access: **write only**
 mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED".

ARGUMENTS

pidadr

VMS Usage: **process_id**
 type: **longword (unsigned)**
 access: **modify**
 mechanism: **by reference**

Process identification number (PID) of the process affected when \$REVOKID completes execution. The ***pidadr*** argument is the address of longword containing the PID of process to be affected. Use -1 to indicate the system rights list. When ***pidadr*** is passed, it is also returned; therefore, you must pass it as a variable rather than a constant.

prcnam

VMS Usage: **process_name**
 type: **character-coded text string**
 access: **read only**
 mechanism: **by descriptor—fixed length string descriptor**

Process name on which \$REVOKID operates. The ***prcnam*** argument is the address of a character string descriptor containing the process name. The maximum length of the name is 15 characters. Since the UIC group number is interpreted as part of the process name, you must use ***pidadr*** to specify the rights list of a process in a different group.

id

VMS Usage: **rights_id**
 type: **quadword (unsigned)**
 access: **modify**
 mechanism: **by reference**

Identifier and attributes to be removed when \$REVOKID completes execution. The ***id*** argument is the address of a quadword containing the binary identifier

System Service Descriptions

\$REVOKID

code to be removed in the first longword and the attributes in the second longword.

Symbol values are offsets to the bits within the longword. You can also obtain the values as masks with the appropriate bit set using the prefix KGB\$M rather than KGB\$V. The symbols are defined in the system macro library (\$KGBDEF).

Bit	Meaning When Set
KGB\$V_DYNAMIC	Allows the unprivileged holder to add or remove the identifier from the process rights list.
KGB\$V_RESOURCE	Allows the holder to charge resources, such as disk blocks, to the identifier.

Either **id** or **name** must be specified. Since the **id** is returned if you specify **name**, as well as passed, you must pass it as a variable rather than a constant in this case.

name

VMS Usage: **char_string**

type: **character-coded text string**

access: **read only**

mechanism: **by descriptor—fixed length string descriptor**

Name of the identifier removed when \$REVOKID completes execution. The **name** argument is the address of a descriptor pointing to the name of the identifier.

prvatr

VMS Usage: **mask_longword**

type: **longword (unsigned)**

access: **write only**

mechanism: **by reference**

Attributes of the deleted identifier. The **prvatr** argument is the address of a longword used to store the attributes of the identifier.

DESCRIPTION

Since the Revoke Identifier from Process service removes the specified identifier from the process's rights list, or from the system rights list, this service is meant for use by a privileged subsystem to alter the access rights profile of a user, based on installation policy. It is not meant for use by the general system user.

CMKRNL privilege is required to invoke this service. In addition, GROUP privilege is required to modify the rights list of a process in the same group as the calling process (unless the process has the same UIC as the calling process). WORLD privilege is required to modify the rights list of a process outside the caller's group. SYSNAM privilege is required to modify the system rights list.

The result of passing the **pidadr** or the **prcnam** argument or both to SYS\$GRANTID is summarized in the following table:

System Service Descriptions

\$REVOKID

prcnam	pidadr	Result
Omitted	Omitted	Current process id is used; process id is not returned.
Omitted	0	Current process id is used; process id is returned.
Omitted	Specified	Specified process id is used; process id is returned.
Specified	Omitted	Specified process name is used; process id is not returned.
Specified	0	Specified process name is used; process id is returned.
Specified	Specified	Specified process id is used; process id is returned; process name is ignored.

The result of passing either the **name** or the **id** argument or both to SY\$GRANTID is summarized in the following table:

name	id	Result
Omitted	Omitted	Illegal
Omitted	Specified	Specified identifier value is used; identifier value is returned.
Specified	Omitted	Specified identifier name is used; identifier value is not returned.
Specified	0	Specified identifier name is used; identifier value is returned.
Specified	Specified	Specified identifier value is used; identifier value is returned; identifier name is ignored.

CONDITION VALUES RETURNED

SS\$_WASCLR	Service is successfully completed; the rights list did not contain the specified identifier.
SS\$_WASSET	Service is successfully completed; the rights list already held the specified identifier.
SS\$_ACCVIO	The pidadr cannot be read or written, or prcnam cannot be read, or id cannot be read or written, or the name cannot be read, or prvatr cannot be written.
SS\$_INSFMEM	Insufficient process dynamic memory is available to open the rights database.
SS\$_NOPRIV	The caller does not have CMKRNL privilege; or is not running in exec or kernel mode; or the caller lacks GROUP, WORLD, or SYSNAM privilege as required.
SS\$_NOSUCHID	The specified identifier name does not exist in the rights database. Note that the binary identifier, if given, is not validated against the rights database.
SS\$_RIGHTSFULL	The process's or system rights list is full.

System Service Descriptions

\$REVOKID

SS\$_IVIDENT

The specified identifier or holder is of invalid format, or the specified identifier and holder are equal.

SS\$_NOSYSNAM

The operation requires SYSNAM privilege.

SS\$_IVLOGNAM

Invalid logical name has been specified.

SS\$_NONEXPR

Nonexistent process has been specified.

RMS\$_PRV

The user does not have read access to the rights database.

Since the rights database is an indexed file accessed with VAX RMS, this service may also return RMS status codes associated with operations on indexed files. Please refer to the *VAX Record Management Services Reference Manual* for descriptions of these status codes.

\$\$SCHDWK—Schedule Wakeup

The Schedule Wakeup service schedules the awakening of a process that has placed itself in a state of hibernation with the Hibernate (\$HIBER) service. A wakeup can be scheduled for a specified absolute time or for a delta time, and can be repeated at fixed intervals.

FORMAT **SYS\$\$SCHDWK** [*pidadr*] , [*prcnam*] , *daytim* , [*reptim*]

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS

pidadr

VMS Usage: **process_id**
type: **longword (unsigned)**
access: **modify**
mechanism: **by reference**

Process identification (PID) of the process that is to be awakened. The **pidadr** argument is the address of a longword containing the PID.

The **pidadr** argument must be specified to awaken processes in other UIC groups.

prcnam

VMS Usage: **process_name**
type: **character-coded text string**
access: **read only**
mechanism: **by descriptor—fixed length string descriptor**

Name of the process to be awakened. The **prcnam** is the address of a character string descriptor pointing to the process name, which is a character string of from 1 to 15 characters.

The **prcnam** argument can be used to awaken only processes in the same UIC group as the calling process. The reason for this is that process names are unique to UIC groups, and VAX/VMS uses the UIC group number of the calling process to interpret the process name specified by the **prcnam** argument. The **pidadr** argument must be used to awaken processes in other UIC groups.

If neither the **pidadr** nor **prcnam** arguments are specified, the wakeup request is issued on behalf of the calling process.

System Service Descriptions

\$SCHDWK

daytim

VMS Usage: **date_time**
type: **quadword (unsigned)**
access: **read only**
mechanism: **by reference**

Time at which the process is to be awakened. The **daytim** argument is the address of a quadword containing this time in the system 64-bit time format. A positive time value specifies an absolute time at which the specified process is to be awakened. A negative time value specifies an offset (delta time) from the current time.

reptim

VMS Usage: **date_time**
type: **quadword (unsigned)**
access: **read only**
mechanism: **by reference**

Time interval at which the wakeup request is to be repeated. The **reptim** argument is the address of a quadword containing this time interval. The time interval must be expressed in delta time format.

The time interval specified cannot be less than 10 milliseconds; if it is less than 10 milliseconds, \$SCHDWK automatically increases it to 10 milliseconds.

If **reptim** is not specified, a default value of 0 is used; zero specifies that the wakeup request is not to be repeated.

DESCRIPTION

Depending on the operation, use of \$SCHDWK may require the calling process to have certain privilege.

- GROUP privilege is required to schedule wake up requests for a process in the same group unless it has the same UIC.
- WORLD privilege is required to schedule wake up requests for any other process in the system.

\$SCHDWK uses the following system resources:

- The AST limit (ASTLM) quota of the calling process is used to schedule a wake up request.
- System dynamic memory is used to allocate a timer queue entry.

If one or more scheduled wake up requests are issued for a process that is not hibernating, a subsequent hibernate request by the target process completes immediately, that is, the process does not hibernate. No count is maintained of outstanding wake up requests.

Scheduled wake up requests that have not yet been processed can be canceled with the Cancel Wakeup (\$CANWAK) service.

If a specified absolute time value has already passed and no repeat time is specified, the timer expires at the next clock cycle (within 10 milliseconds).

System Service Descriptions

\$SCHDWK

CONDITION VALUES RETURNED

SS\$_NORMAL

Service successfully completed.

SS\$_ACCVIO

The expiration time, repeat time, process name string or string descriptor cannot be read by the caller, or the process identification cannot be written by the caller.

SS\$_EXQUOTA

The process has exceeded its AST limit quota.

SS\$_INSFMEM

Insufficient system dynamic memory is available to allocate a timer queue entry.

SS\$_JVLOGNAM

The process name string has a length of 0 or has more than 15 characters.

SS\$_IVTIME

The specified delta repeat time is a positive value, or an absolute time plus delta repeat time is less than the current time.

SS\$_NONEXPR

Warning. The specified process does not exist, or an invalid process identification was specified.

SS\$_NOPRIV

The process does not have the privilege to schedule a wake up request for the specified process.

System Service Descriptions

\$SETAST

\$SETAST—Set AST Enable

The Set AST Enable service enables or disables the delivery of ASTs for the access mode from which the service call was issued.

FORMAT **SYS\$SETAST** *enbflg*

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENT *enbflg*

VMS Usage: **Boolean**
type: **byte (unsigned)**
access: **read only**
mechanism: **by value**

Value specifying whether ASTs are to be enabled. The **enbflg** argument is a byte containing this value. A value of 1 enables AST delivery for the calling access mode; a value of 0 disables AST delivery.

DESCRIPTION When an image is executing in user mode, ASTs are enabled for all higher access modes.

If ASTs are disabled for a more privileged access mode, VAX/VMS cannot deliver ASTs for less privileged access modes until ASTs are enabled once again for the more privileged access mode. Therefore, a process that has disabled ASTs for a more privileged access mode must re-enable ASTs for that mode before returning to a less privileged access mode.

**CONDITION
VALUES
RETURNED**

SS\$_WASCLR

Service successfully completed. AST delivery was previously disabled for the calling access mode.

SS\$_WASSET

Service successfully completed. AST delivery was previously enabled for the calling access mode.

\$SETEF—Set Event Flag

The Set Event Flag service sets an event flag in a local or common event flag cluster. The condition value returned by \$SETEF indicates whether the specified flag was previously set or clear. Once the event flag is set, processes waiting for the event flag to be set resume execution.

FORMAT **SYS\$SETEF** *efn*

RETURNS

VMS Usage: **cond_value**
 type: **longword (unsigned)**
 access: **write only**
 mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENT *efn*

VMS Usage: **ef_number**
 type: **longword (unsigned)**
 access: **read only**
 mechanism: **by value**

Number of the event flag to be set. The *efn* argument is a longword containing this number.

There are two local event flag clusters which are local to the process: cluster 0 and cluster 1. Cluster 0 contains event flag numbers 0 to 31, and cluster 1 contains event flag numbers 32 to 63.

There are two common event flag clusters: cluster 2 and cluster 3. Cluster 2 contains event flag numbers 64 to 95, and cluster 3 contains event flag numbers 96 to 127.

**CONDITION
VALUES
RETURNED**

SS\$_WASCLR
 SS\$_WASSET
 SS\$_ILLEFC
 SS\$_UNASEFC

Service successfully completed. The specified event flag was previously 0.

Service successfully completed. The specified event flag was previously 1.

An illegal event flag number was specified.

The process is not associated with the cluster containing the specified event flag.

System Service Descriptions

\$SETEXV

\$SETEXV—Set Exception Vector

The Set Exception Vector service (1) assigns a condition handler address to the primary, secondary, or last chance exception vectors or (2) removes a previously assigned handler address from any of these three vectors.

FORMAT **SYS\$SETEXV** [*vector*],[*addres*],[*acmode*],[*prvhnd*]

RETURNS VMS Usage: **cond_value**
 type: **longword (unsigned)**
 access: **write only**
 mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS **vector**
 VMS Usage: **longword_unsigned**
 type: **longword (unsigned)**
 access: **read only**
 mechanism: **by value**

Vector for which a condition handler is to be established or removed. The **vector** argument is a longword value. A value of 0 (the default) specifies the primary vector; a value of 1, the secondary vector; and a value of 2, the last chance exception vector.

addres
VMS Usage: **procedure**
type: **procedure entry mask**
access: **call without stack unwinding**
mechanism: **by reference**

Condition handler address to be established for the exception vector specified by **vector**. The **addres** argument is a longword value containing the address of the entry mask to the condition handler routine.

If **addres** is not specified or is specified as 0, the condition handler address already established for the specified vector is removed; that is, the contents of the longword vector is set to 0.

acmode
VMS Usage: **access_mode**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Access mode for which the exception vector is to be modified. The **acmode** argument is a longword containing the access mode. The \$PSLDEF macro defines symbols for the four access modes.

System Service Descriptions

\$SETEXV

The most privileged access mode used is the access mode of the caller. Exception vectors for access modes more privileged than the caller's access mode cannot be modified.

prvhnd

VMS Usage: **procedure**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Previous condition handler address contained by the specified exception vector. The **prvhnd** argument is the address of a longword into which \$SETEF writes the handler address.

DESCRIPTION

A process cannot modify a vector associated with a more privileged access mode.

VAX/VMS provides two different methods for establishing condition handlers:

- Using the call stack associated with each access mode. Each call frame includes a longword to contain the address of a condition handler associated with that frame. The RTL routine LIB\$ESTABLISH establishes a condition handler; the RTL routine LIB\$REVERT removes a handler.
- Using the software exception vectors (by using \$SETEXV) associated with each access mode. These vectors are set aside in the process's control region (P1 space).

The second method does not possess the modular properties associated with the first method. The software exception vectors are intended for use primarily for performance monitors and debuggers. For example, the primary exception vector and the last chance exception vector are used by the VAX Debugger for user mode access, and DCL uses the last chance exception vector for supervisor mode access.

User mode exception vectors are canceled at image exit.

CONDITION VALUES RETURNED

SS\$_NORMAL
SS\$_ACCVIO

Service successfully completed.

The longword that is to receive the previous contents of the vector cannot be written by the caller.

System Service Descriptions

\$SETIME

\$SETIME—Set System Time

The Set System Time service (1) changes the value of or (2) recalibrates the system time.

The system time is defined by a quadword value that specifies the number of 100 nanosecond intervals since 00:00 o'clock, November 17, 1858.

The system time is the reference used for nearly all timer-related software activities in VAX/VMS.

FORMAT	SYS\$SETIME [<i>timadr</i>]
---------------	--------------------------------------

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENT	<i>timadr</i>
-----------------	----------------------

VMS Usage: **date_time**
type: **quadword (unsigned)**
access: **read only**
mechanism: **by reference**

New time value for the system time. The *timadr* argument is the address of a quadword containing the new system time value. The new system time value is an absolute time value specifying the number of 100 nanosecond intervals since 00:00 o'clock, November 17, 1858. A negative (delta) time value is invalid.

If *timadr* is not specified or is specified as 0, \$SETIME recalibrates the system time using the time-of-year clock.

DESCRIPTION	To set the system time, the calling process must have OPER and LOG_IO privileges.
--------------------	---

After changing or recalibrating the system clock, \$SETIME updates the timer queue by adjusting each element in the timer queue by the difference between the previous system time and the new system time.

The \$SETIME service saves the new time (for future bootstrap operations) in the system image SYS\$SYSTEM:SYS.EXE. To save the time, the service assigns a channel to the system boot device and calls the \$QIOW service. This I/O operation requires the LOG_IO user privilege.

System Service Descriptions

\$SETIME

CONDITION VALUES RETURNED

SS\$_NORMAL
SS\$_IVTIME
SS\$_ACCVIO
SS\$_NOIOCHAN
SS\$_NOPRIV

Service successfully completed.

The caller specified no time value or a negative time value and an invalid processor clock was found.

The quadword that contains the new system time value cannot be read by the caller.

No I/O channel is available for assignment.

The process does not have the privileges to set the system time.

System Service Descriptions

\$SETIMR

\$SETIMR—Set Timer

The Set Timer service sets the timer to expire at a specified time. When the timer expires, an event flag is set and (optionally) an AST routine executes.

FORMAT **SYS\$SETIMR** [*efn*] ,*daytim* ,[*astadr*] ,[*reqidt*]

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS *efn*

VMS Usage: **ef_number**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Event flag to be set when the timer expires. The *efn* argument is a longword value containing the number of the event flag. If *efn* is not specified, event flag 0 is set.

When \$SETIMR first executes, it clears the specified event flag or event flag 0.

daytim

VMS Usage: **date_time**
type: **quadword (unsigned)**
access: **read only**
mechanism: **by reference**

Time at which the timer expires. The *daytim* argument is the address of a quadword time value. A positive time value specifies an absolute time at which the timer expires; a negative time value specifies an offset (delta time) from the current time.

If a specified absolute time value has already passed, the timer expires at the next clock cycle, which would be within 10 milliseconds.

The Convert ASCII String to Binary Time (\$BINTIM) service converts an ASCII string time value to the quadword time value required by \$SETIMR.

astadr

VMS Usage: **ast_procedure**
type: **procedure entry mask**
access: **call without stack unwinding**

System Service Descriptions

\$SETIMR

mechanism: **by reference**

AST service routine that is to execute when the timer expires. The **astadr** is the address of the entry mask of this routine. If **astadr** is not specified or is specified as 0 (the default), no AST routine executes.

The AST routine, if specified, executes at the access mode of the caller.

reqidt

VMS Usage: **user_arg**

type: **longword (unsigned)**

access: **read only**

mechanism: **by value**

Identification of the timer request. The **reqidt** is a longword value containing a number that uniquely identifies the timer request. If **reqidt** is not specified, the value 0 is used.

To cancel a timer request, the identification of the timer request (as specified by **reqidt** in \$SETIMR) is passed to the Cancel Timer (\$CANTIM) service (as the **reqidt** argument).

If cancellation of specific timer requests, but not all timer requests, is desired, be sure to specify a nonzero value for **reqidt** in the \$SETIMR call because \$CANTIM interprets an identification value of zero as a request to cancel all timer requests.

Unique values for **reqidt** can be specified for each timer request, or the same value can be given to related timer requests. This allows for selective cancelling of a single timer request, a group of related timer requests, or all timer requests.

If the **astadr** argument is specified in the \$SETIMR call, the value specified by the **reqidt** argument is passed as a parameter to the AST routine. If the AST routine requires more than one parameter, specify an address for the value of **reqidt**; the AST routine can then interpret that address as the beginning of a list of parameters.

DESCRIPTION

The Set Timer service requires dynamic memory and uses the process's timer queue entries (TQELM) quota. If an AST routine is specified, the service uses the process's AST limit (ASTLM) quota.

The \$SETIMR service executes at the access mode of the caller, as does the AST routine, if one is specified.

CONDITION VALUES RETURNED

SS\$_NORMAL
SS\$_ACCVIO
SS\$_EXQUOTA

Service successfully completed.

The expiration time cannot be read by the caller.

The process exceeded its quota for timer entries or its AST limit quota; or there is insufficient system dynamic memory to complete the request.

SS\$_ILLEFC
SS\$_INSFMEM

An illegal event flag number was specified.

Insufficient dynamic memory is available to allocate a timer queue entry.

SS\$_UNASEFC

The process is not associated with the cluster containing the specified event flag.

System Service Descriptions

\$SETPRA

\$SETPRA—Set Power Recovery AST

The Set Power Recovery AST service establishes a routine to receive control after a power recovery is detected.

FORMAT **SYS\$SETPRA** *astadr*, [*acmode*]

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS *astadr*

VMS Usage: **ast_procedure**
type: **procedure entry mask**
access: **call without stack unwinding**
mechanism: **by reference**

Power recovery AST routine to receive control when a power recovery is detected. The *astadr* is the address of the entry mask of this routine.

If *astadr* specifies 0, an AST is not delivered to the process when a power recovery is detected.

The system passes one parameter to the specified AST routine. This parameter is a longword value containing the length of time that the power was off, expressed as the number of 1/100th of a second intervals that have elapsed.

acmode

VMS Usage: **access_mode**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Access mode at which the power recovery AST routine is to execute. The *acmode* argument is a longword containing the access mode. The \$PSLDEF macro defines symbols for the four access modes.

The most privileged access mode used is the access mode of the caller.

System Service Descriptions

\$SETPRA

DESCRIPTION The \$SETPRA system service uses the process's AST limit (ASTLM) quota. Only one power recovery AST routine can be specified for a process. The AST entry point address is cleared at image exit.

The entry and exit conventions for the power recovery AST routine are the same as for all AST service routines. These conventions are described in Section 5.

**CONDITION
VALUES
RETURNED**

SS\$_NORMAL
SS\$_EXQUOTA

Service successfully completed.
The process exceeded its quota for outstanding
AST requests.

System Service Descriptions

\$SETPRI

\$SETPRI—Set Priority

The Set Priority service changes a process's base priority. The base priority is used to determine the order in which executable processes are to run.

FORMAT **SY\$SETPRI** [*pidadr*],[*prcnam*],*pri*,[*prvpri*]

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS *pidadr*

VMS Usage: **process_id**
type: **longword (unsigned)**
access: **modify**
mechanism: **by reference**

Process identification (PID) of the process whose priority is to be set. The *pidadr* argument is the address of the PID.

prcnam

VMS Usage: **process_name**
type: **character-coded text string**
access: **read only**
mechanism: **by descriptor—fixed length string descriptor**

Process name of the process whose priority is to be changed. The *prcnam* argument is the address of a character string descriptor pointing to a 1- to 15-character process name string.

The *prcnam* argument can be used only on behalf of processes in the same UIC group as the calling process. To set the priority for processes in other groups, the *pidadr* argument must be specified.

If neither the *pidadr* nor *prcnam* arguments are specified, \$SETPRI sets the calling process's base priority.

pri

VMS Usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

New base priority to be established for the process. The *pri* argument is a longword value containing the new priority. Priorities which are not real time are in the range 0 through 15; real-time priorities are in the range 16 through 31.

System Service Descriptions

\$SETPRI

If the specified priority is higher than the base priority of the target process, and if the caller does not have ALTPRI privilege, then the base priority of the target process is used.

prvpri

VMS Usage: **longword_unsigned**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Base priority of the process before the call to \$SETPRI. The *prvpri* argument is the address of a longword into which \$SETPRI writes the process's previous base priority.

DESCRIPTION

Depending on the operation, use of \$SETPRI may require the calling process to have one of the privileges listed below.

- GROUP privilege is required to change the priority of a process in the same group, unless the target process has the same UIC as the calling process.
- WORLD privilege is required to change the priority of any other process in the system.
- ALTPRI privilege is required to set any process's priority to a value greater than the target process's initial base priority.

A process's base priority remains in effect until specifically changed or until the process is deleted.

If a process does not have ALTPRI privilege and attempts to set a priority higher than the base priority of the target process, the priority is set to the base priority of the target process, and the status code SS\$_NORMAL is returned.

To determine the priority set by the \$SETPRI service, use the Get Job/Process Information (\$GETJPI) service.

CONDITION VALUES RETURNED

SS\$_NORMAL	Service successfully completed.
SS\$_ACCVIO	The process name string or string descriptor cannot be read by the caller, or the process identification or previous priority longword cannot be written by the caller.
SS\$_IVLOGNAM	The process name string has a length of 0 or has more than 15 characters.
SS\$_NONEXPR	Warning. The specified process does not exist, or an invalid process identification was specified.
SS\$_NOPRIV	The process does not have the privilege to affect other processes.

System Service Descriptions

\$SETPRN

\$SETPRN—Set Process Name

The Set Process Name service allows a process to establish or to change its own process name.

FORMAT **SYS\$SETPRN** [*prcnam*]

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENT

prcnam

VMS Usage: **process_name**
type: **character-coded text string**
access: **read only**
mechanism: **by descriptor—fixed length string descriptor**

Process name to be given to the calling process. The **prcnam** argument is the address of a character string descriptor pointing to a 1- to 15-character process name string. If **prcnam** is not specified, the calling process is given no name.

DESCRIPTION

A process name remains in effect until it is changed (using \$SETPRN) or until the process is deleted.

Process names provide an identification mechanism for processes executing with the same group number. A process can also be identified by its process identification (PID).

**CONDITION
VALUES
RETURNED**

SS\$_NORMAL

Service successfully completed.

SS\$_ACCVIO

The process name string or string descriptor cannot be read by the caller.

SS\$_DUPLNAM

The specified process name duplicates one already specified within that group.

SS\$_JVLOGNAM

The specified process name has a length of 0 or has more than 15 characters.

\$SETPRT—Set Protection on Pages

The Set Protection on Pages service allows a process to change the protection on a page or range of pages.

FORMAT

SYS\$SETPRT *inadr* ,*[retadr]* ,*[acmode]* ,*prot* ,*[prvppt]*

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS *inadr*

VMS Usage: **address_range**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Starting and ending virtual addresses of the range of pages whose protection is to be changed. The *inadr* is the address of a two-longword array containing, in order, the starting and the ending process virtual addresses. Only the virtual page number portion of each virtual address is used; the low-order 9 bits are ignored.

If the starting and ending virtual addresses are the same, the protection is changed for a single page.

retadr

VMS Usage: **address_range**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference—array reference or descriptor**

Starting and ending virtual addresses of the range of pages whose protection was actually changed by \$SETPRT. The *retadr* is the address of a two-longword array containing, in order, the starting and ending process virtual addresses.

If an error occurs while the protection is being changed, \$SETPRT writes into *retadr* the range of pages that were successfully changed before the error occurred. If no pages were affected before the error occurred, \$SETPRT writes the value -1 into each longword of the two-longword array.

acmode

VMS Usage: **access_mode**
type: **longword (unsigned)**
access: **read only**

System Service Descriptions

\$SETPRT

mechanism: **by value**

Access mode associated with the call to \$SETPRT. The **acmode** argument is a longword containing the access mode. The \$PSLDEF macro defines symbols for the four access modes.

\$SETPRT uses whichever of the following two access modes is least privileged: (1) the access mode specified by **acmode** or (2) the access mode of the caller. To change the protection of any page in the specified range, the resultant access mode must be equal to or more privileged than the access mode of the owner of that page.

prot

VMS Usage: **page_protection**

type: **longword (unsigned)**

access: **read only**

mechanism: **by value**

Page protection to be assigned to the specified pages. The **prot** argument is a longword value containing the protection code. Only bits 0 to 3 are used; bits 4 to 31 are ignored.

The \$PRTDEF macro defines the following symbolic names for the protection codes:

Symbol	Description
PRT\$_NA	No access
PRT\$_KR	Kernel read only
PRT\$_KW	Kernel write
PRT\$_ER	Executive read only
PRT\$_EW	Executive write
PRT\$_SR	Supervisor read only
PRT\$_SW	Supervisor write
PRT\$_UR	User read only
PRT\$_UW	User write
PRT\$_ERKW	Executive read; kernel write
PRT\$_SRKW	Supervisor read; kernel write
PRT\$_SREW	Supervisor read; executive write
PRT\$_URKW	User read; kernel write
PRT\$_UREW	User read; executive write
PRT\$_URSW	User read; supervisor write

If the protection is specified as 0, the protection defaults to kernel read-only.

prvpri

VMS Usage: **page_protection**

type: **byte (unsigned)**

access: **write only**

mechanism: **by reference**

Protection previously assigned to the last page in the range. The **prvpri** argument is the address of a byte into which \$SETPRT writes the protection of this page. The **prvpri** argument is useful only when protection for a single page is being changed.

System Service Descriptions

\$SETPRT

DESCRIPTION If a process changes any pages in a private section from read-only to read/write, \$SETPRT uses the process's paging file (PGFLQUOTA) quota.

For pages in global sections, the new protection can alter only copy-on-reference pages.

CONDITION VALUES RETURNED

SS\$_NORMAL

Service successfully completed.

SS\$_ACCVIO

The input address array cannot be read by the caller; the output address array or the byte to receive the previous protection cannot be written by the caller; or an attempt was made to change the protection of a nonexistent page.

SS\$_EXQUOTA

The process exceeded its paging file quota while changing a page in a read-only private section to a read/write page.

SS\$_IVPROTECT

The specified protection code has a numeric value of 1 or is greater than 15.

SS\$_LENVIO

A page in the specified range is beyond the end of the program or control region.

SS\$_NOPRIV

A page in the specified range is in the system address space.

SS\$_PAGOWNVIO

Page owner violation. An attempt was made to change the protection on a page owned by a more privileged access mode.

System Service Descriptions

\$SETPRV

\$SETPRV—Set Privileges

The Set Privileges service enables or disables specified privileges for the calling process.

FORMAT **SYS\$SETPRV** [*enbflg*] , [*prvadr*] , [*prmflg*] , [*prvprv*]

RETURNS VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS ***enbflg***

VMS Usage: **Boolean**
type: **byte (unsigned)**
access: **read only**
mechanism: **by value**

Indicator specifying whether the specified privileges are to be enabled or disabled. The ***enbflg*** is a byte value. A value of 1 specifies that the privileges specified in the ***prvadr*** argument are to be enabled. A value of 0 (the default) indicates that the privileges are to be disabled.

prvadr

VMS Usage: **mask_privileges**
type: **quadword (unsigned)**
access: **read only**
mechanism: **by reference**

Privileges to be enabled or disabled for the calling process. The ***prvadr*** argument is the address of a quadword bit vector wherein each bit corresponds to a privilege that is to be enabled or disabled.

Each bit has a symbolic name; these names are defined by the \$PRVDEF macro. The bit vector is formed by specifying the symbolic name of each desired privilege in a logical OR operation. Table SYS-5 gives the symbolic name and description of each privilege.

Table SYS-5 User Privileges

Privilege	Symbolic name	Description
ALLSPOOL	PRV\$V_ALLSPOOL	Allocate a spooled device
BUGCHK	PRV\$V_BUGCHK	Make bugcheck error log entries
BYPASS	PRV\$V_BYPASS	Bypass UIC-based protection
CMEXEC	PRV\$V_CMEXEC	Change mode to executive
CMKRNL	PRV\$V_CMKRNL	Change mode to kernel

System Service Descriptions

\$SETPRV

Table SYS-5 (Cont.) User Privileges

Privilege	Symbolic name	Description
DETACH	PRV\$_DETACH	Create detached processes
DIAGNOSE	PRV\$_DIAGNOSE	May diagnose devices
DOWNGRADE	PRV\$_DOWNGRADE	May downgrade classification
EXQUOTA	PRV\$_EXQUOTA	May exceed quotas
GROUP	PRV\$_GROUP	Group process control
GRPNAM	PRV\$_GRPNAM	Place name in group logical name table
GRPPRV	PRV\$_GRPPRV	Group access via system protection field
LOG_IO	PRV\$_LOG_IO	Perform logical I/O operations
MOUNT	PRV\$_MOUNT	Issue mount volume QIO
NETMBX	PRV\$_NETMBX	Create a network device
ACNT	PRV\$_NOACNT	Create processes for which no accounting is done
OPER	PRV\$_OPER	All operator privileges
PFNMAP	PRV\$_PFNMAP	Map to section by physical page frame number
PHY_IO	PRV\$_PHY_IO	Perform physical I/O operations
PRMCEB	PRV\$_PRMCEB	Create permanent common event flag clusters
PRMGBL	PRV\$_PRMGBL	Create permanent global sections
PRMJNL	PRV\$_PRMJNL	May create permanent journals
PRMMBX	PRV\$_PRMMBX	Create permanent mailboxes
PSWAPM	PRV\$_PSWAPM	Change process swap mode
READALL	PRV\$_READALL	Possess read access to everything
SECURITY	PRV\$_SECURITY	May perform security functions
ALTPRI	PRV\$_SETPRI	Set (alter) any process priority
SETPRV	PRV\$_SETPRV	Set any process privileges
SHARE	PRV\$_SHARE	May assign a channel to a non-shared device
SHMEM	PRV\$_SHMEM	Allocate structures in memory shared by multiple processors
SYSGBL	PRV\$_SYSGBL	Create system global sections
SYSLCK	PRV\$_SYSLCK	Queue system-wide locks
SYSNAM	PRV\$_SYSNAM	Place name in system logical name table
SYSPRV	PRV\$_SYSPRV	Access files and other resources as if you have a system UIC
TMPJNL	PRV\$_TMPJNL	May create temporary journals
TMPMBX	PRV\$_TMPMBX	Create temporary mailboxes
UPGRADE	PRV\$_UPGRADE	May upgrade classification

System Service Descriptions

\$SETPRV

Table SYS-5 (Cont.) User Privileges

Privilege	Symbolic name	Description
VOLPRO	PRV\$V_VOLPRO	Override volume protection
WORLD	PRV\$V_WORLD	World process control

Note that the names of the privilege bits PRV\$V_NOACNT and PRV\$V_SETPRI correspond to the names of the DCL privileges ACNT and ALTPRI, yet have different names.

If the **prvadr** is not specified or is specified as 0, the privileges are not altered.

prmflg

VMS Usage: **Boolean**
type: **byte (unsigned)**
access: **read only**
mechanism: **by value**

Indicator specifying whether the privileges are to be affected permanently or temporarily. The **prmflg** is a byte value. A value of 1 specifies that the privileges are to be affected permanently, that is, until they are again changed by using \$SETPRV or until the process is deleted. A value of 0 (the default) specifies that the privileges are to be affected temporarily, that is, until the current image exits (at which time the process's permanently enabled privileges will be restored).

prvprv

VMS Usage: **mask_privileges**
type: **quadword (unsigned)**
access: **write only**
mechanism: **by reference**

Privileges previously possessed by the calling process. The **prvprv** argument is the address of a quadword bit vector wherein each bit corresponds to a privilege that was previously either enabled or disabled. If **prvprv** is not specified or is specified as 0, the previous privilege mask is not returned.

DESCRIPTION

To set a privilege permanently, the calling process must be authorized to set the specified privilege, or the process must be executing in kernel or executive mode.

To set a privilege temporarily, one of the three following conditions must be true:

- The calling process must be authorized to set the specified privilege.
- The calling process must be executing in kernel or executive mode.
- The image currently executing must be an image that was installed with the specified privilege.

VAX/VMS maintains four separate privilege masks for each process:

- AUTHPRIV—Privileges that the process is authorized to enable, as designated by the system manager or the process creator. The AUTHPRIV mask never changes during the life of the process.
- PROCPRIV—Privileges that are designated as permanently enabled for the process. The PROCPRIV mask can be modified by \$SETPRV.

System Service Descriptions

\$SETPRV

- IMAGPRIV—Privileges that the current image is installed with.
- CURPRIV—Privileges that are currently enabled. The CURPRIV mask can be modified by \$SETPRV.

When a process is created, its AUTHPRIV, PROCPRIV, and CURPRIV masks have the same contents. Whenever a system service (other than \$SETPRV) must check the process privileges, that service checks the CURPRIV mask.

When a process runs an installed image, the privileges with which that image was installed are enabled in the CURPRIV mask. When the installed image exits, the PROCPRIV mask is copied to the CURPRIV mask.

The \$SETPRV service can set bits only in the CURPRIV and PROCPRIV masks, but \$SETPRV checks the AUTHPRIV mask to see whether a process can set specified privilege bits in the CURPRIV or PROCPRIV masks. Consequently, a process can give itself the SETPRV privilege only if this privilege is enabled in the AUTHPRIV mask.

Each of a process's four privilege masks may be obtained by calling the Get Job/Process Information (\$GETJPI) service and specifying the desired privilege mask(s) as item code(s) in the *itmlst* argument. The item code for a privilege mask is constructed by prefixing the name of the privilege mask with the characters *JPI\$*— (for example, *JPI\$_CURPRIV* is the item code for the current privilege mask).

The DCL command SET PROCESS/PRIVILEGES also enables or disables specified privileges; refer to the *VAX/VMS DCL Dictionary* for details.

CONDITION VALUES RETURNED

SS\$_NORMAL	Service successfully completed. All privileges were enabled or disabled as specified.
SS\$_NOTALLPRIV	Service successfully completed; not all specified privileges were enabled; see the Description section for details.
SS\$_ACCVIO	The privilege mask cannot be read or the previous privilege mask cannot be written by the caller.

System Service Descriptions

\$SETRWM

\$SETRWM—Set Resource Wait Mode

The Set Resource Wait Mode service allows a process to specify what action system services should take when system resources required for their execution are unavailable.

When resource wait mode is enabled, system services will wait for the required system resources to become available and then will continue execution.

When resource wait mode is disabled, system services will return to the caller when required system resources are unavailable.

The condition value returned by \$SETRWM indicates whether resource wait mode was previously enabled or previously disabled.

FORMAT	SYS\$SETRWM [<i>watflg</i>]
---------------	--------------------------------------

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENT

watflg

VMS Usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Indicator specifying whether system services should wait for required resources. The **watflg** argument is a longword value. A value of 0 (the default) specifies that system services should wait until resources needed for their execution become available. A value of 1 specifies that system services should return failure status immediately when resources needed for their execution are unavailable.

VAX/VMS enables resource wait mode for all processes. Resource wait mode can be disabled only by calling \$SETRWM.

If resource wait mode is disabled, it remains disabled until it is explicitly reenabled or until the process is deleted.

System Service Descriptions

\$SETRWM

DESCRIPTION The following system resources and process quotas are affected by resource wait mode:

- System dynamic memory
- UNIBUS adapter map registers
- Direct I/O limit (DIOLM) quota
- Buffered I/O limit (BIOLM) quota
- Buffered I/O byte count limit (BYTLM) quota

**CONDITION
VALUES
RETURNED**

SS\$_WASCLR

Service successfully completed. Resource wait mode was previously enabled.

SS\$_WASSET

Service successfully completed. Resource wait mode was previously disabled.

System Service Descriptions

\$SETSFM

\$SETSFM—Set System Service Failure Exception Mode

The Set System Service Failure Exception Mode service allows a process to specify whether or not VMS should generate a software exception when a system service returns an error or severe error condition value to the calling process.

The \$SETSFM indicates in the condition value it returns whether system service exception mode was enabled or disabled previous to the call (to \$SETSFM).

Initially, system service failure exception mode is disabled, so the caller should explicitly test for successful completion following a system service call.

FORMAT

SYS\$SETSFM [*enbflg*]

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENT

enbflg

VMS Usage: **Boolean**
type: **byte (unsigned)**
access: **read only**
mechanism: **by value**

Number specifying whether the system service failure exception mode is to be enabled. The ***enbflg*** argument is a byte value. A value of 1 specifies that the system service failure exception mode is enabled. A value of 0 (the default) specifies that the system service failure exception mode is disabled.

System Service Descriptions

\$SETSFM

DESCRIPTION

When enabled, a software exception is generated when a system service returns an error or severe error condition value. System service failure exceptions are generated only if the service call originated from user mode. The \$SETSFM service can be called, however, from any access mode.

If enabled, system service failure exception mode remains enabled until explicitly disabled or until the image exits. A condition handler can be specified in the first longword of the procedure call stack or with the Set Exception Vector (\$SETEXV) service. If no condition handler is specified by the user, a default system handler is used. This condition handler causes the image to exit and then displays the exit status.

The argument list provided to the condition handler contains the code SS\$_SSFAIL in the condition name argument of the signal array.

For an explanation and examples of condition handling routines, the format of the argument lists passed to the condition handler, and a discussion of the appropriate actions a condition handler may take, see Section 10.

CONDITION VALUES RETURNED

SS\$_WASCLR

Service successfully completed. Failure exceptions were previously disabled.

SS\$_WASSET

Service successfully completed. Failure exceptions were previously enabled.

System Service Descriptions

\$SETSSF

\$SETSSF—Set System Services Filter

The Set System Services Filter service inhibits user-mode calls to certain system services. The following system services cannot be inhibited by \$SETSSF:

\$ASCTIM	\$BINTIM	\$EXIT	\$FAO
\$FAOL	\$PUTMSG	\$UNWIND	

FORMAT **SYS\$SETSSF** [*mask*]

RETURNS VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS **mask**

VMS Usage: **mask_longword**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Category of system services that are to be inhibited for user-mode calls. The **mask** argument is a longword value of which only the first byte is significant. The first byte is a bit vector wherein a bit when set specifies a category of system service to be inhibited. Only bits 0 and 1 are used; bits 2 to 7 are reserved.

When bit 0 is set, all system services, including user-written system services, are inhibited except those listed prior to the Format section.

When bit 1 is set, all system services, including user-written system services, are inhibited except those listed prior to the Format section and those listed below:

\$ADJSTK \$CRETVA \$DELTVA \$GETMSG
\$SETSFM

Bit 1 inhibits fewer system services than bit 0. Specifically, bit 1 does not inhibit the system services required by condition-handling and image-rundown services, whereas bit 0 does.

System Service Descriptions

\$SETSSF

DESCRIPTION

Successfully calling \$SETSSF requires that the access mode of the caller be equal to or more privileged than supervisor mode access and that the SYSGEN parameter SSINHIBIT be set when the system is bootstrapped.

If a system service that has been inhibited is called from user-mode, one of the following two condition values is returned:

SS\$_INHCHME	A disabled executive mode system service was called.
SS\$_INHCHMK	A disabled kernel mode system service was called.

CONDITION VALUES RETURNED

SS\$_NORMAL	Service successfully completed.
SS\$_NOPRIV	The process does not have the privilege to call the service.

System Service Descriptions

\$SETSTK

\$SETSTK—Set Stack Limits

The Set Stack Limits (\$SETSTK) service allows a process to change the size of its supervisor, executive, and kernel stacks by altering the values in the stack limit and base arrays held in P1 (per-process) space.

FORMAT

SYS\$SETSTK *inadr* [,*retadr*] [,*acmode*]

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS *inadr*

VMS Usage: **address_range**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Range of addresses that express the stack's new limits. The *inadr* argument is the address of a two-longword array containing, in order, the address of the top of the stack and the address of the base of the stack. Since stacks in P1 space expand from high to low addresses, the address of the base of the stack must be greater than the address of the top of the stack.

retadr

VMS Usage: **address_range**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Range of addresses that express the stack's previous limits. The *retadr* argument is the address of a two-longword array into which \$SETSTK writes, in the first longword, the previous address of the top of the stack and, in the second longword, the previous address of the base of the stack.

acmode

VMS Usage: **access_mode**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Access mode of the stack to be altered. The *acmode* argument is a longword containing the access mode. The \$PSLDEF macro defines symbols for the four access modes. The most privileged access mode used is the access mode of the caller.

If *acmode* specifies user mode, \$SETSTK performs no operation and returns the SS\$_NORMAL condition value.

System Service Descriptions

\$SETSTK

DESCRIPTION	The calling process can only adjust the size of stacks for access modes that are equal to or less privileged than the access mode of the calling process.
--------------------	---

CONDITION VALUES RETURNED	SS\$_NORMAL SS\$_ACCVIO	Service successfully completed. The input address array cannot be read by the caller; the input range is invalid; or the return address array cannot be written by the caller.
--	----------------------------	---

System Service Descriptions

\$SETSWM

\$SETSWM—Set Process Swap Mode

The Set Process Swap Mode service allows a process to control whether or not it can be swapped out of the balance set.

When process swap mode is enabled, the process can be swapped out; when disabled, the process remains in the balance set until (1) process swap mode is re-enabled or (2) the process is deleted.

The \$SETSWM service returns a condition value indicating whether process swap mode was enabled or disabled previous to the call to \$SETSWM.

To lock some but not necessarily all process pages into the balance set, use the Lock Pages in Memory (\$LCKPAG) service.

FORMAT	SYS\$SETSWM [<i>swpflg</i>]
---------------	--------------------------------------

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENT

swpflg

VMS Usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Indicator specifying whether or not the process can be swapped. The **swpflg** is a longword value. A value of 0 (the default) enables process swap mode, meaning the process can be swapped. A value of 1 disables process swap mode, meaning the process cannot be swapped.

DESCRIPTION	To change its process swap mode, the calling process must have PSWAPM privilege.
--------------------	--

CONDITION VALUES RETURNED

SS\$_WASCLR

Service successfully completed. The process was not previously locked in the balance set.

SS\$_WASSET

Service successfully completed. The process was previously locked in the balance set.

SS\$_NOPRIV

The process does not have the necessary PSWAPM privilege.

\$SETUAI—Set User Authorization Information

The Set User Authorization Information (\$SETUAI) service is used to modify the user authorization file (UAF) record for a specified user.

FORMAT

SYS\$SETUAI *[nullarg],[nullarg],usrnam,itmlst
[nullarg],[nullarg]*

RETURNS

VMS Usage: cond_value
type: longword (unsigned)
access: write only
mechanism: by value

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS

nullarg

VMS Usage: **null_arg**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Place-holding argument. This argument is reserved to DIGITAL.

usrnam

VMS Usage: **char_string**
type: **character-coded text string**
access: **read only**
mechanism: **by descriptor-fixed length string descriptor**

Name of the user whose user authorization file (UAF) record is modified. The **usrnam** argument is the address of a descriptor pointing to a character text string containing the user name. The user name string may contain a maximum of 12 alphanumeric characters.

itmlst

```

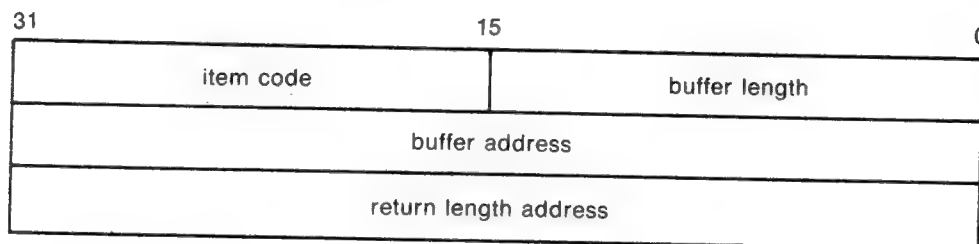
VMS Usage:  item_list_3
type:       longword (unsigned)
access:     read only
mechanism:  by reference

```

Item list specifying which information from the specified user's UAF (user authorization file) record is to be modified. The **itemlist** argument is the address of a list of one or more item descriptors, each of which specifies an item code. The item list is terminated by an item code of 0 or by a longword of 0. The following diagram depicts the structure of a single item descriptor.

System Service Descriptions

\$SETUAI



ZK-1705-84

\$SETUAI Item Descriptor Fields

buffer length

A word specifying the length (in bytes) of the buffer in which \$SETUAI is to write the information. The length of the buffer varies depending on the item code specified in the **item code** field of the item descriptor and is given in the description of each item code. If the value of **buffer length** is too small, \$SETUAI truncates the data.

item code

A word containing a user-supplied symbolic code specifying the item of information that \$SETUAI is to set. These codes are defined by the \$UAIDF macro and have the format: `UAI$_code`. Each item code is described below.

buffer address

A longword containing the user-supplied address of the buffer in which \$SETUAI is to write the information.

return length address

A longword containing the user-supplied address of a word in which \$SETUAI writes the length in bytes of the information it actually set.

\$SETUAI Item Codes

UAI\$_ACCOUNT

When `UAI$_ACCOUNT` is specified, \$SETUAI sets, as a blank-filled character string, the account name of the user. Since an account name can include up to 8 characters plus a size-byte prefix, the buffer length field of the item descriptor should specify 9 (bytes).

UAI\$_ASTLM

When `UAI$_ASTLM` is specified, \$SETUAI sets the AST queue limit, which is a longword integer in the range 1 through 65,535.

UAI\$_BATCH_ACCESS_P

When `UAI$_BATCH_ACCESS_P` is specified, \$SETUAI sets the range of times during which batch access is permitted for primary days. Each bit set in the longword represents a one-hour period, from bit 0 as midnight to 1 AM to bit 23 as 11 PM to midnight.

UAI\$_BATCH_ACCESS_S

When `UAI$_BATCH_ACCESS_S` is specified, \$SETUAI sets the range of times during which batch access is permitted for secondary days. Each bit set in the longword represents a one-hour period, from bit 0 as midnight to 1 AM to bit 23 as 11 PM to midnight.

System Service Descriptions

\$SETUAI

UAI\$_BIOLM

When UAI\$_BIOLM is specified, \$SETUAI sets the buffered I/O count limit, which is a longword integer in the range 1 through 65,535.

UAI\$_BYTLM

When UAI\$_BYTLM is specified, \$SETUAI sets the buffered I/O byte limit, which is a longword integer in the range 1 through 65,535.

UAI\$_CLITABLES

When UAI\$_CLITABLES is specified, \$SETUAI sets, as a character string, the name of the user-defined CLI table for the account, if any. Since the CLI table name can include up to 31 characters plus a size-byte prefix, the buffer length field of the item descriptor should specify 32 (bytes).

UAI\$_CPUTIM

When UAI\$_CPUTIM is specified, \$SETUAI sets, as a longword decimal value, the maximum CPU time limit (per session) for the process in 10-millisecond units.

UAI\$_DEFCLI

When UAI\$_DEFCLI is specified, \$SETUAI sets, as an RMS file name component, the name of the command language interpreter used to execute the specified batch job. The file specification set assumes the device name and directory SYS\$SYSTEM: and the file type EXE. Since a file name can include up to 39 characters plus a size-byte prefix, the buffer length field in the item descriptor should specify 40 (bytes).

UAI\$_DEFDEV

When UAI\$_DEFDEV is specified, \$SETUAI sets, as a 1- to 15-character string, the name of the default device. Since the device name string can include up to 15 characters plus a size-byte prefix, the buffer length field in the item descriptor should specify 16 (bytes).

UAI\$_DEFDIR

When UAI\$_DEFDIR is specified, \$SETUAI sets, as a 1- to 63-character string, the name of the default directory. Since the directory name string can include up to 63 characters plus a size-byte prefix, the buffer length field in the item descriptor should specify 64 (bytes).

UAI\$_DEF_PRIV

When UAI\$_DEF_PRIV is specified, \$SETUAI sets, as a quadword value, the default privileges for the user.

UAI\$_DFWSCNT

When UAI\$_DFWSCNT is specified, \$SETUAI sets the default working set size, which is a longword integer in the range 1 through 65,535.

UAI\$_DIOLM

When UAI\$_DIOLM is specified, \$SETUAI sets the direct I/O count limit, which is a longword integer in the range 1 through 65,535.

UAI\$_DIALUP_ACCESS_P

When UAI\$_DIALUP_ACCESS_P is specified, \$SETUAI sets the range of times during which dialup access is permitted for primary days. Each bit set in the longword represents a one-hour period, from bit 0 as midnight to 1 AM to bit 23 as 11 PM to midnight.

System Service Descriptions

\$SETUAI

UAI\$_DIALUP_ACCESS_S

When UAI\$_DIALUP_ACCESS_S is specified, \$SETUAI sets the range of times during which dialup access is permitted for secondary days. Each bit set in the longword represents a one-hour period, from bit 0 as midnight to 1 AM to bit 23 as 11 PM to midnight.

UAI\$_ENQLM

When UAI\$_ENQLM is specified, \$SETUAI sets the lock queue limit, which is a longword integer in the range 1 through 65,535.

UAI\$_EXPIRATION

When UAI\$_EXPIRATION is specified, \$SETUAI sets, as a quadword absolute time value, the expiration date and time of the account.

UAI\$_FILLM

When UAI\$_FILLM is specified, \$SETUAI sets the open file limit, which is a longword integer in the range 1 through 65,535.

UAI\$_FLAGS

When UAI\$_FLAGS is specified, \$SETUAI sets, as a longword bit vector, the various login flags set for the user. Each flag is represented by a bit. Listed below are the symbolic names for these flags, which are defined by the \$UAIDEF macro.

Symbol	Description
UAI\$_AUDIT	All actions are audited.
UAI\$_CAPTIVE	User is restricted to captive account.
UAI\$_DEFCLI	User is restricted to default command interpreter.
UAI\$_DISCTLY	User cannot use CTRL/Y.
UAI\$_DISMAIL	Announcement of new mail is suppressed.
UAI\$_DISRECONNECT	User cannot reconnect to existing processes.
UAI\$_DISREPORT	User will not receive last login messages.
UAI\$_DISWELCOME	User will not receive the login welcome message.
UAI\$_GENPWD	User is required to use generated passwords.
UAI\$_LOCKPWD	SET PASSWORD command is disabled.
UAI\$_NOMAIL	Mail delivery to user is disabled.
UAI\$_PWD_EXPIRED	Primary password is expired.
UAI\$_PWD2_EXPIRED	Secondary password is expired.

UAI\$_JTQUOTA

When UAI\$_JTQUOTA is specified, \$SETUAI sets the initial byte quota with which the jobwide logical name table is to be created, which is a longword integer in the range 1 through 65,365.

UAI\$_LGICMD

When UAI\$_LGICMD is specified, \$SETUAI sets, as an RMS file specification, the name of the default login command file. Since a file specification can include up to 255 characters plus a size-byte prefix, the buffer length field of the item descriptor should specify 256 (bytes).

System Service Descriptions

\$SETUAI

UAI\$_LOCAL_ACCESS_P

When **UAI\$_LOCAL_ACCESS_P** is specified, **\$SETUAI** sets the range of times during which local interactive access is permitted for primary days. Each bit set in the longword represents a one-hour period, from bit 0 as midnight to 1 AM to bit 23 as 11 PM to midnight.

UAI\$_LOCAL_ACCESS_S

When **UAI\$_LOCAL_ACCESS_S** is specified, **\$SETUAI** sets the range of times during which local interactive access is permitted for secondary days. Each bit set in the longword represents a one-hour period, from bit 0 as midnight to 1 AM to bit 23 as 11 PM to midnight.

UAI\$_MAXACCTJOBS

When **UAI\$_MAXACCTJOBS** is specified, **\$SETUAI** sets, as a longword integer, the maximum number of batch, interactive, and detached processes which may be active at one time for all users of the same account. A value of 0 represents an unlimited number.

UAI\$_MAXDETACH

When **UAI\$_MAXDETACH** is specified, **\$SETUAI** sets the detached process limit, which is a longword integer in the range 1 through 65,535. A value of 0 represents an unlimited number.

UAI\$_MAXJOBS

When **UAI\$_MAXJOBS** is specified, **\$SETUAI** sets the active process limit, which is a longword integer in the range 1 through 65,535. A value of 0 represents an unlimited number.

UAI\$_NETWORK_ACCESS_P

When **UAI\$_NETWORK_ACCESS_P** is specified, **\$SETUAI** sets the range of times during which network access is permitted for primary days. Each bit set in the longword represents a one-hour period, from bit 0 as midnight to 1 AM to bit 23 as 11 PM to midnight.

UAI\$_NETWORK_ACCESS_S

When **UAI\$_NETWORK_ACCESS_S** is specified, **\$SETUAI** sets the range of times during which network access is permitted for secondary days. Each bit set in the longword represents a one-hour period, from bit 0 as midnight to 1 AM to bit 23 as 11 PM to midnight.

UAI\$_OWNER

When **UAI\$_OWNER** is specified, **\$SETUAI** sets, as a character string, the name of the owner of the account. Since the owner name can include up to 31 characters plus a size-byte prefix, the buffer length field of the item descriptor should specify 32 (bytes).

UAI\$_PBYTLM

When **UAI\$_PBYTLM** is specified, **\$SETUAI** sets the paged buffer I/O byte count limit, which is a longword integer in the range 1 through 65,535.

UAI\$_PGFLQUOTA

When **UAI\$_PGFLQUOTA** is specified, **\$SETUAI** sets the paging file quota, which is a longword integer in the range 1 through 65,535.

UAI\$_PRCCNT

When **UAI\$_PRCCNT** is specified, **\$SETUAI** sets the subprocess creation limit, which is a longword integer in the range 1 through 65,535.

System Service Descriptions

\$SETUAI

UAI\$_PRI

When UAI\$_PRI is specified, \$SETUAI sets the default base priority, which is a longword integer in the range 0 through 31.

UAI\$_PRIMEDAYS

When UAI\$_PRIMEDAYS is specified, \$SETUAI sets, as a longword bit vector, the primary and secondary days of the week. Each bit represents a day of the week, with the bit clear representing a primary day and the bit set representing a secondary day. Listed below are the symbolic names for these bits, which are defined by the \$UAIDEF macro.

UAI\$_V_MONDAY
UAI\$_V_TUESDAY
UAI\$_V_WEDNESDAY
UAI\$_V_THURSDAY
UAI\$_V_FRIDAY
UAI\$_V_SATURDAY
UAI\$_V_SUNDAY

UAI\$_PRIV

When UAI\$_PRIV is specified, \$SETUAI sets, as a quadword value, the names of the privileges held by the user.

UAI\$_PWD

When UAI\$_PWD is specified, \$SETUAI sets, as a quadword value, the hashed primary password of the user.

UAI\$_PWD_LENGTH

When UAI\$_PWD_LENGTH is specified, \$SETUAI sets the minimum password length, which is a longword integer.

UAI\$_PWD_LIFETIME

When UAI\$_PWD_LIFETIME is specified, \$SETUAI sets, as a quadword absolute time value, the password lifetime.

UAI\$_PWD2

When UAI\$_PWD2 is specified, \$SETUAI sets, as a quadword value, the hashed secondary password of the user.

UAI\$_QUEPRI

When UAI\$_QUEPRI is specified, \$SETUAI sets the maximum job queue priority, which is a longword integer in the range 0 through 31.

UAI\$_REMOTE_ACCESS_P

When UAI\$_REMOTE_ACCESS_P is specified, \$SETUAI sets the range of times during which batch access is permitted for primary days. Each bit set in the longword represents a one-hour period, from bit 0 as midnight to 1 AM to bit 23 as 11 PM to midnight.

UAI\$_REMOTE_ACCESS_S

When UAI\$_REMOTE_ACCESS_S is specified, \$SETUAI sets the range of times during which batch access is permitted for secondary days. Each bit set in the longword represents a one-hour period, from bit 0 as midnight to 1 AM to bit 23 as 11 PM to midnight.

UAI\$_SHRFILLM

When UAI\$_SHRFILLM is specified, \$SETUAI sets the shared file limit, which is a longword integer in the range 1 through 65,535.

System Service Descriptions

\$SETUAI

UAI\$_TQCNT

When UAI\$_TQCNT is specified, \$SETUAI sets the timer queue entry limit, which is a longword integer in the range 1 through 65,535.

UAI\$_UIC

When UAI\$_UIC is specified, \$SETUAI sets, as a longword, the user identification code (UIC), containing the following two word-length subfields:

Symbolic Name	Description
UIC\$W_MEM	The member number subfield of the UIC
UIC\$W_GRP	The group number subfield of the UIC

UAI\$_USERNAME

When UAI\$_USERNAME is specified, \$SETUAI sets, as a blank-filled character string of up to 12 bytes, the username of the owner of the specified job.

UAI\$_WSEXTENT

When UAI\$_WSEXTENT is specified, \$SETUAI sets the working set extent specified for the specified job or queue, which is a longword integer in the range 1 through 65,535.

UAI\$_WSQUOTA

When UAI\$_WSQUOTA is specified, \$SETUAI sets the working set quota for the specified user, which is a longword integer in the range 1 through 65,535.

DESCRIPTION

Use the following list to determine the privileges required to use the \$SETUAI service:

- BYPASS or SYSPRV—allows modification of any record in the UAF (user authorization file)
- GRPPRV—allows modification of any record in the UAF whose UIC group matches that of the requester. A group manager with GRPPRV privilege is limited in the extent to which he may modify the UAF records of users in the same group; values such as privileges and quotas may only be changed if the modification does not exceed the values set in the group manager's UAF record.
- No privilege—does not allow access to any UAF record

System Service Descriptions

\$SETUAI

CONDITION VALUES RETURNED

SS\$_NORMAL

Successful completion.

SS\$_ACCVIO

The item list or input buffer cannot be read by the caller; or the return length buffer, output buffer, or status block cannot be written by the caller.

SS\$_BADPARAM

The function code is invalid; the item list contains an invalid item code; a buffer descriptor has an invalid length; or the reserved parameter has a nonzero value.

SS\$_NOPRIV

The user does not have the privileges required to examine the authorization information for the specified user.

\$SNDDERR—Send Message to Error Logger

The Send Message to Error Logger service writes a user-specified message to the system error log file, preceding it with the date and time.

FORMAT **SY\$SNDDERR** *msgbuf*

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED".

ARGUMENT

msgbuf

VMS Usage: **char_string**
type: **character-coded text string**
access: **read only**
mechanism: **by descriptor—fixed length string descriptor**

Message to be written to the error log file. The *msgbuf* argument is the address of a character string descriptor pointing to the message text.

DESCRIPTION

To send a message to the error log file, the calling process must have BUGCHK privilege.

The \$SNDDERR service requires system dynamic memory.

**CONDITION
VALUES
RETURNED**

SS\$_NORMAL
SS\$_ACCVIO
SS\$_INSFMEM
SS\$_NOPRIV

Service successfully completed.

The message buffer or buffer descriptor cannot be read by the caller.

Insufficient system dynamic memory is available to complete the service.

The process does not have the required BUGCHK privilege.

System Service Descriptions

\$SNDJBC

\$SNDJBC—Send to Job Controller

The Send to Job Controller (\$SNDJBC) service creates, stops, and manages queues and the batch and print jobs in those queues. See the Description section for a discussion of the types of queue supported by the VAX/VMS batch/print facility. The \$SNDJBC and \$GETQUI services together provide the user interface to the VAX/VMS Job Controller, which is the VAX/VMS queue and accounting manager.

The \$SNDJBC service completes asynchronously, that is, it returns to the caller after queuing the request, without waiting for the operation to complete.

To synchronize the completion of most operations, use the Send to Job Controller and Wait (\$SNDJBCW) service. The \$SNDJBCW service is identical to \$SNDJBC in every way except that \$SNDJBCW returns to the caller after the operation has completed.

For additional information about system service completion, refer to the Synchronize (\$SYNCH) service and to Section 2.5.

The \$SNDJBC and \$SNDJBCW services supersede the Send Message to Symbiont Manager (\$SND SMB) and Send Message to Accounting Manager (\$SND ACC) services. New programs should be written using \$SNDJBC or \$SNDJBCW, instead of \$SND SMB or \$SND ACC, and old programs using \$SND SMB or \$SND ACC should be converted to use \$SNDJBC or \$SNDJBCW as convenient.

FORMAT	SY\$SNDJBC [<i>efn</i>], <i>func</i> [, <i>nullarg</i>] [, <i>itmlst</i>] [, <i>iosb</i>] [, <i>astadr</i>] [, <i>astprm</i>]
---------------	---

RETURNS	VMS Usage: cond_value type: longword (unsigned) access: write only mechanism: by value
----------------	---

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS	<i>efn</i> VMS Usage: ef_number type: longword (unsigned) access: read only mechanism: by value
------------------	--

Number of the event flag to be set when \$SNDJBC completes. The *efn* argument is a longword containing this number. The *efn* argument is optional.

System Service Descriptions

\$SNDJBC

When the request is queued, \$SNDJBC clears the specified event flag (or event flag 0 if **efn** was not specified). Then when the operation has completed, \$SNDJBC sets the specified event flag (or event flag 0).

func

VMS Usage: **function_code**
type: **word (unsigned)**
access: **read only**
mechanism: **by value**

Function code specifying the function that \$SNDJBC is to perform. The **func** argument is a word containing this function code. The \$SJCDEF macro defines the names of each function code.

Only one function code may be specified in a single call to \$SNDJBC. Most function codes require or allow for additional information to be passed in the call. This information is passed by using the **itmlst** argument, which specifies a list of one or more item descriptors. Each item descriptor in turn specifies an item code, which modifies, restricts, or otherwise affects the action designated by the function code.

The following lists each function code, describes the action it designates, and lists which item code(s) must and may be specified; descriptions of the item codes appear in the description of the **itmlst** argument:

\$SNDJBC Function Codes With Their Valid Item Codes

SJC\$_ABORT_JOB

This request aborts execution of a current job. The job can be deleted, or it can be requeued if it is restartable.

The following input item code must be specified:

SJC\$_QUEUE

The following input item code must be specified for batch jobs:

SJC\$_ENTRY_NUMBER

The following input or Boolean item codes may be specified:

SJC\$_DESTINATION_QUEUE

SJC\$_HOLD

SJC\$_NO_HOLD

SJC\$_PRIORITY

SJC\$_REQUEUE

SJC\$_ADD_FILE

This request adds a file to the open job owned by the requesting process. You use this operation as part of a sequence of calls to the \$SNDJBC service to create a job with one or more files. The first call in the sequence specifies the SJC\$_CREATE_JOB operation to create an open job. Each subsequent SJC\$_ADD_FILE request associates an additional file with the job. Finally, you make a SJC\$_CLOSE_JOB request to complete the batch or print job specification. To create a job that contains only one file, you can make a single call to \$SNDJBC that specifies the SJC\$_ENTER_FILE function code.

One of the following input item codes must be specified:

SJC\$_FILE_IDENTIFICATION

SJC\$_FILE_SPECIFICATION

System Service Descriptions

\$SNDJBC

The following input or Boolean item codes may be specified:

SJC\$_DELETE_FILE	SJC\$_NO_DELETE_FILE
SJC\$_DOUBLE_SPACE	SJC\$_NO_DOUBLE_SPACE
SJC\$_FILE_BURST	SJC\$_NO_FILE_BURST
SJC\$_FILE_COPIES	
SJC\$_FILE_FLAG	SJC\$_NO_FILE_FLAG
SJC\$_FILE_SETUP_MODULES	SJC\$_NO_FILE_SETUP_MODULES
SJC\$_FILE_TRAILER	SJC\$_NO_FILE_TRAILER
SJC\$_FIRST_PAGE	SJC\$_NO_FIRST_PAGE
SJC\$_LAST_PAGE	SJC\$_NO_LAST_PAGE
SJC\$_PAGE_HEADER	SJC\$_NO_PAGE_HEADER
SJC\$_PAGINATE	SJC\$_NO_PAGINATE
SJC\$_PASSALL	SJC\$_NO_PASSALL

SJC\$_ALTER_JOB

This request alters the parameters of an existing job that is not currently executing.

The following input item codes must be specified:

SJC\$_QUEUE
SJC\$_ENTRY_NUMBER

The following input or Boolean item codes may be specified:

SJC\$_AFTER_TIME	SJC\$_NO_AFTER_TIME
SJC\$_CHARACTERISTIC_NAME	SJC\$_NO_CHARACTERISTICS
SJC\$_CHARACTERISTIC_NUMBER	
	SJC\$_NO_CHECKPOINT_DATA
SJC\$_CLI	SJC\$_NO_CLI
SJC\$_CPU_LIMIT	SJC\$_NO_CPU_LIMIT
	SJC\$_NO_DELETE_FILE
SJC\$_DESTINATION_QUEUE	
SJC\$_DOUBLE_SPACE	SJC\$_NO_DOUBLE_SPACE
SJC\$_FILE_BURST	SJC\$_NO_FILE_BURST
SJC\$_FILE_COPIES	
SJC\$_FILE_FLAG	SJC\$_NO_FILE_FLAG
SJC\$_FILE_SETUP_MODULES	SJC\$_NO_FILE_SETUP_MODULES
SJC\$_FILE_TRAILER	SJC\$_NO_FILE_TRAILER
SJC\$_FIRST_PAGE	SJC\$_NO_FIRST_PAGE
SJC\$_FORM_NAME	
SJC\$_FORM_NUMBER	
SJC\$_HOLD	SJC\$_NO_HOLD
SJC\$_JOB_COPIES	
SJC\$_JOB_NAME	

System Service Descriptions

SSNDJBC

SJC\$_LAST_PAGE	SJC\$_NO_LAST_PAGE
SJC\$_LOG_DELETE	SJC\$_NO_LOG_DELETE
SJC\$_LOG_QUEUE	
SJC\$_LOG_SPECIFICATION	SJC\$_NO_LOG_SPECIFICATION
SJC\$_LOG_SPOOL	SJC\$_NO_LOG_SPOOL
SJC\$_LOWERCASE	SJC\$_NO_LOWERCASE
SJC\$_NOTE	SJC\$_NO_NOTE
SJC\$_NOTIFY	SJC\$_NO_NOTIFY
SJC\$_OPERATOR_REQUEST	SJC\$_NO_OPERATOR_REQUEST
SJC\$_PAGE_HEADER	SJC\$_NO_PAGE_HEADER
SJC\$_PAGINATE	SJC\$_NO_PAGINATE
SJC\$_PARAMETER_1 through 8	SJC\$_NO_PARAMETERS
SJC\$_PASSALL	SJC\$_NO_PASSALL
SJC\$_PRIORITY	
SJC\$_RESTART	SJC\$_NO_RESTART
SJC\$_WSDEFAULT	SJC\$_NO_WSDEFAULT
SJC\$_WSEXTENT	SJC\$_NO_WSEXTENT
SJC\$_WSQUOTA	SJC\$_NO_WSQUOTA

SJC\$_ALTER_QUEUE

This request alters the parameters of a queue. The execution of current jobs is unaffected.

The following input item code must be specified:

SJC\$_QUEUE

The following input or Boolean item codes may be specified:

SJC\$_BASE_PRIORITY	
SJC\$_CHARACTERISTIC_NAME	SJC\$_NO_CHARACTERISTICS
SJC\$_CHARACTERISTIC_NUMBER	
SJC\$_CPU_DEFAULT	SJC\$_NO_CPU_DEFAULT
SJC\$_CPU_LIMIT	SJC\$_NO_CPU_LIMIT
SJC\$_DEFAULT_FORM_NAME	
SJC\$_DEFAULT_FORM_NUMBER	
SJC\$_FILE_BURST	SJC\$_NO_FILE_BURST
SJC\$_FILE_BURST_ONE	
SJC\$_FILE_FLAG	SJC\$_NO_FILE_FLAG
SJC\$_FILE_FLAG_ONE	
SJC\$_FILE_TRAILER	SJC\$_NO_FILE_TRAILER
SJC\$_FILE_TRAILER_ONE	
SJC\$_FORM_NAME	
SJC\$_FORM_NUMBER	
SJC\$_GENERIC_SELECTION	SJC\$_NO_GENERIC_SELECTION

System Service Descriptions

\$SNDJBC

SJC\$_JOB_BURST	SJC\$_NO_JOB_BURST
SJC\$_JOB_FLAG	SJC\$_NO_JOB_FLAG
SJC\$_JOB_LIMIT	
SJC\$_JOB_RESET_MODULES	SJC\$_NO_JOB_RESET_MODULES
SJC\$_JOB_SIZE_MAXIMUM	SJC\$_NO_JOB_SIZE_MAXIMUM
SJC\$_JOB_SIZE_MINIMUM	SJC\$_NO_JOB_SIZE_MINIMUM
SJC\$_JOB_SIZE_SCHEDULING	SJC\$_NO_JOB_SIZE_SCHEDULING
SJC\$_JOB_TRAILER	SJC\$_NO_JOB_TRAILER
SJC\$_OWNER_UIC	
SJC\$_PAGINATE	SJC\$_NO_PAGINATE
SJC\$_PROTECTION	
SJC\$_RECORD_BLOCKING	SJC\$_NO_RECORD_BLOCKING
SJC\$_RETAIN_ALL_JOBS	SJC\$_NO_RETAIN_JOBS
SJC\$_RETAIN_ERROR_JOBS	
SJC\$_SWAP	SJC\$_NO_SWAP
SJC\$_WSDEFAULT	SJC\$_NO_WSDEFAULT
SJC\$_WSEXTENT	SJC\$_NO_WSEXTENT
SJC\$_WSQUOTA	SJC\$_NO_WSQUOTA

SJC\$_ASSIGN_QUEUE

This request assigns a logical queue to an execution queue. The logical queue is specified by the SJC\$_QUEUE item code; the execution queue, by the SJC\$_DESTINATION_QUEUE item code.

The following input item codes must be specified:

SJC\$_QUEUE
SJC\$_DESTINATION_QUEUE

SJC\$_BATCH_CHECKPOINT

This request establishes a checkpoint in a batch job. No operation is performed if the requesting process is not a batch process.

The following input item code must be specified:

SJC\$_CHECKPOINT_DATA

SJC\$_CLOSE_DELETE

This request deletes the open job owned by the requesting process. No item codes are allowed.

SJC\$_CLOSE_JOB

This request completes the specification of the open job owned by the requesting process and places the job in the queue specified in the SJC\$_CREATE_JOB request that opened the job. If the SJC\$_CLOSE_JOB request completes successfully, the job is no longer an open job; it becomes a normal batch or print job.

The following output item code may be specified:

SJC\$_JOB_STATUS_OUTPUT

System Service Descriptions

\$SNDJBC

SJC\$_CREATE_JOB

This request creates an open job for the requesting process. If the process already owns an open job, that job is deleted.

An open job is a batch or print job that has not yet been completely specified. After you make the SJC\$_CREATE_JOB request to open the job, you can make subsequent calls to \$SNDJBC using the SJC\$_ADD_FILE function code to specify the files associated with the job. Finally, you can complete the job specification with an SJC\$_CLOSE_JOB request. If the SJC\$_CREATE_JOB operation completes successfully, the open job created is given an entry number; the job is not assigned to the queue specified in the SJC\$_CREATE_JOB operation until the SJC\$_CLOSE_JOB successfully completes.

The following input item code must be specified:

SJC\$_QUEUE

The following input or Boolean item codes may be specified:

SJC\$_ACCOUNT_NAME	
SJC\$_AFTER_TIME	SJC\$_NO_AFTER_TIME
SJC\$_CHARACTERISTIC_NAME	SJC\$_NO_CHARACTERISTICS
SJC\$_CHARACTERISTIC_NUMBER	
SJC\$_CLI	SJC\$_NO_CLI
SJC\$_CPU_LIMIT	SJC\$_NO_CPU_LIMIT
SJC\$_FILE_BURST	SJC\$_NO_FILE_BURST
SJC\$_FILE_BURST_ONE	
SJC\$_FILE_FLAG	SJC\$_NO_FILE_FLAG
SJC\$_FILE_FLAG_ONE	
SJC\$_FILE_TRAILER	SJC\$_NO_FILE_TRAILER
SJC\$_FILE_TRAILER_ONE	
SJC\$_FORM_NAME	
SJC\$_FORM_NUMBER	
SJC\$_HOLD	SJC\$_NO_HOLD
SJC\$_JOB_COPIES	
SJC\$_JOB_NAME	
SJC\$_LOG_DELETE	SJC\$_NO_LOG_DELETE
SJC\$_LOG_QUEUE	
SJC\$_LOG_SPECIFICATION	SJC\$_NO_LOG_SPECIFICATION
SJC\$_LOG_SPOOL	SJC\$_NO_LOG_SPOOL
SJC\$_LOWERCASE	SJC\$_NO_LOWERCASE
SJC\$_NOTE	SJC\$_NO_NOTE
SJC\$_NOTIFY	SJC\$_NO_NOTIFY
SJC\$_OPERATOR_REQUEST	SJC\$_NO_OPERATOR_REQUEST
SJC\$_PARAMETER_1 through 8	SJC\$_NO_PARAMETERS
SJC\$_PRIORITY	

System Service Descriptions

\$SNDJBC

SJC\$_RESTART	SJC\$_NO_RESTART
SJC\$_UIC	
SJC\$_USERNAME	
SJC\$_WSDEFAULT	SJC\$_NO_WSDEFAULT
SJC\$_WSEXTENT	SJC\$_NO_WSEXTENT
SJC\$_WSQUOTA	SJC\$_NO_WSQUOTA

The following output item code may be specified:

SJC\$_ENTRY_NUMBER_OUTPUT

SJC\$_CREATE_QUEUE

This request creates a queue. If the queue already exists and is not stopped, this request performs no operation. However, if the queue already exists and is stopped, the request alters the parameters of the queue based on the item codes specified in the request; if the SJC\$_CREATE_START item code is specified, the request starts the queue.

The following input item code must be specified:

SJC\$_QUEUE

The following input or Boolean item codes may be specified:

SJC\$_BASE_PRIORITY	
SJC\$_BATCH	SJC\$_NO_BATCH
SJC\$_CHARACTERISTIC_NAME	SJC\$_NO_CHARACTERISTICS
SJC\$_CHARACTERISTIC_NUMBER	
SJC\$_CPU_DEFAULT	SJC\$_NO_CPU_DEFAULT
SJC\$_CPU_LIMIT	SJC\$_NO_CPU_LIMIT
SJC\$_CREATE_START	
SJC\$_DEFAULT_FORM_NAME	
SJC\$_DEFAULT_FORM_NUMBER	
SJC\$_DEVICE_NAME	
SJC\$_FILE_BURST	SJC\$_NO_FILE_BURST
SJC\$_FILE_BURST_ONE	
SJC\$_FILE_FLAG	SJC\$_NO_FILE_FLAG
SJC\$_FILE_FLAG_ONE	
SJC\$_FILE_TRAILER	SJC\$_NO_FILE_TRAILER
SJC\$_FILE_TRAILER_ONE	
SJC\$_FORM_NAME	
SJC\$_FORM_NUMBER	
SJC\$_GENERIC_QUEUE	SJC\$_NO_GENERIC_QUEUE
SJC\$_GENERIC_SELECTION	SJC\$_NO_GENERIC_SELECTION
SJC\$_GENERIC_TARGET	
SJC\$_JOB_BURST	SJC\$_NO_JOB_BURST
SJC\$_JOB_FLAG	SJC\$_NO_JOB_FLAG

System Service Descriptions

\$SNDJBC

SJC\$_JOB_LIMIT	
SJC\$_JOB_RESET_MODULES	SJC\$_NO_JOB_RESET_MODULES
SJC\$_JOB_SIZE_MAXIMUM	SJC\$_NO_JOB_SIZE_MAXIMUM
SJC\$_JOB_SIZE_MINIMUM	SJC\$_NO_JOB_SIZE_MINIMUM
SJC\$_JOB_SIZE_SCHEDULING	SJC\$_NO_JOB_SIZE_SCHEDULING
SJC\$_JOB_TRAILER	SJC\$_NO_JOB_TRAILER
SJC\$_LIBRARY_SPECIFICATION	SJC\$_NO_LIBRARY_SPECIFICATION
SJC\$_OWNER_UIC	
SJC\$_PAGINATE	SJC\$_NO_PAGINATE
SJC\$_PROCESSOR	SJC\$_NO_PROCESSOR
SJC\$_PROTECTION	
SJC\$_RECORD_BLOCKING	SJC\$_NO_RECORD_BLOCKING
SJC\$_RETAIN_ALL_JOBS	SJC\$_NO_RETAIN_JOBS
SJC\$_RETAIN_ERROR_JOBS	
SJC\$_SCSNODE_NAME	
SJC\$_SWAP	SJC\$_NO_SWAP
SJC\$_TERMINAL	SJC\$_NO_TERMINAL
SJC\$_WSDEFAULT	SJC\$_NO_WSDEFAULT
SJC\$_WSEXTENT	SJC\$_NO_WSEXTENT
SJC\$_WSQUOTA	SJC\$_NO_WSQUOTA

SJC\$_DEASSIGN_QUEUE

This request deassigns a logical queue from an execution queue.

The following input item code must be specified:

SJC\$_QUEUE

SJC\$_DEFINE_CHARACTERISTIC

This request defines a characteristic name and number and inserts this definition in the queue file. Each characteristic name must have a unique number in the range 0 to 127. If the characteristic name is already defined, the request alters the definition of the characteristic.

A job cannot execute on an execution queue unless the queue possesses all the characteristics possessed by the job; the queue may possess additional characteristics and the job will still execute.

The following input item codes must be specified:

SJC\$_CHARACTERISTIC_NAME
SJC\$_CHARACTERISTIC_NUMBER

SJC\$_DEFINE_FORM

This request defines a form name and number, as well as other physical attributes of the paper stock used in printers, and inserts this definition into the system job queue file. If the form name is already defined, this request alters the definition of the form.

System Service Descriptions

\$SNDJBC

Forms are used only by output execution queues and print jobs. A print job cannot execute unless the stock name of a form specified for the queue is the same as the stock name specified for the job. The stock name of a form, which is specified by using the `SJC$_FORM_STOCK` item code, specifies the paper stock used by the printer. Other item codes specify printing parameters for a job such as the margins, length of paper, and so on.

Each form must have a unique number. When a new queue file is created, the system supplies the definition of a form named `DEFAULT` with number 0 and default characteristics.

The following input item codes must be specified:

`SJC$_FORM_NAME`
`SJC$_FORM_NUMBER`

The following input or Boolean item codes may be specified:

<code>SJC\$_FORM_DESCRIPTION</code>	
<code>SJC\$_FORM_LENGTH</code>	
<code>SJC\$_FORM_MARGIN_BOTTOM</code>	
<code>SJC\$_FORM_MARGIN_LEFT</code>	
<code>SJC\$_FORM_MARGIN_RIGHT</code>	
<code>SJC\$_FORM_MARGIN_TOP</code>	
<code>SJC\$_FORM_SETUP_MODULES</code>	<code>SJC\$_NO_FORM_SETUP_MODULES</code>
<code>SJC\$_FORM_SHEET_FEED</code>	<code>SJC\$_NO_FORM_SHEET_FEED</code>
<code>SJC\$_FORM_STOCK</code>	
<code>SJC\$_FORM_TRUNCATE</code>	<code>SJC\$_NO_FORM_TRUNCATE</code>
<code>SJC\$_FORM_WIDTH</code>	
<code>SJC\$_FORM_WRAP</code>	<code>SJC\$_NO_FORM_WRAP</code>
<code>SJC\$_PAGE_SETUP_MODULES</code>	<code>SJC\$_NO_PAGE_SETUP_MODULES</code>

SJC\$_DELETE_CHARACTERISTIC

This request deletes the definition of a characteristic name.

The following input item code must be specified:

`SJC$_CHARACTERISTIC_NAME`

SJC\$_DELETE_FORM

This request deletes the definition of a form name. There must be no queues or jobs that reference the form.

The following input item code must be specified:

`SJC$_FORM_NAME`

SJC\$_DELETE_JOB

This request deletes a job from the system job queue file. If the job is currently executing, it is aborted.

The following input item codes must be specified:

`SJC$_QUEUE`
`SJC$_ENTRY_NUMBER`

System Service Descriptions

\$SNDJBC

SJC\$_DELETE_QUEUE

This request deletes a queue and all of the jobs in the queue. The queue must be stopped, and there must be no other queues or jobs that reference the queue.

The following input item code must be specified:

SJC\$_QUEUE

SJC\$_ENTER_FILE

This request creates a job containing one file and places the job in the specified queue. To create a job with more than one file, you must make a sequence of calls to the \$SNDJBC service using the SJC\$_CREATE_JOB, SJC\$_ADD_FILE, and SJC\$_CLOSE_JOB function codes.

The following input item code must be specified:

SJC\$_QUEUE

One of the following input item codes must be specified:

SJC\$_FILE_IDENTIFICATION

SJC\$_FILE_SPECIFICATION

The following input or Boolean item codes may be specified:

SJC\$_ACCOUNT_NAME

SJC\$_AFTER_TIME

SJC\$_CHARACTERISTIC_NAME

SJC\$_CHARACTERISTIC_NUMBER

SJC\$_CLI

SJC\$_CPU_LIMIT

SJC\$_DELETE_FILE

SJC\$_DOUBLE_SPACE

SJC\$_FILE_BURST

SJC\$_FILE_COPIES

SJC\$_FILE_FLAG

SJC\$_FILE_SETUP_MODULES

SJC\$_FILE_TRAILER

SJC\$_FIRST_PAGE

SJC\$_FORM_NAME

SJC\$_FORM_NUMBER

SJC\$_HOLD

SJC\$_JOB_COPIES

SJC\$_JOB_NAME

SJC\$_LAST_PAGE

SJC\$_LOG_DELETE

SJC\$_LOG_QUEUE

SJC\$_LOG_SPECIFICATION

SJC\$_LOG_SPOOL

SJC\$_NO_AFTER_TIME

SJC\$_NO_CHARACTERISTICS

SJC\$_NO_CLI

SJC\$_NO_CPU_LIMIT

SJC\$_NO_DELETE_FILE

SJC\$_NO_DOUBLE_SPACE

SJC\$_NO_FILE_BURST

SJC\$_NO_FILE_FLAG

SJC\$_NO_FILE_SETUP_MODULES

SJC\$_NO_FILE_TRAILER

SJC\$_NO_FIRST_PAGE

SJC\$_NO_HOLD

SJC\$_NO_LAST_PAGE

SJC\$_NO_LOG_DELETE

SJC\$_NO_LOG_SPECIFICATION

SJC\$_NO_LOG_SPOOL

System Service Descriptions

\$SNDJBC

SJC\$_LOWERCASE	SJC\$_NO_LOWERCASE
SJC\$_NOTE	SJC\$_NO_NOTE
SJC\$_NOTIFY	SJC\$_NO_NOTIFY
SJC\$_OPERATOR_REQUEST	SJC\$_NO_OPERATOR_REQUEST
SJC\$_PAGE_HEADER	SJC\$_NO_PAGE_HEADER
SJC\$_PAGINATE	SJC\$_NO_PAGINATE
SJC\$_PARAMETER_1 through 8	SJC\$_NO_PARAMETERS
SJC\$_PASSALL	SJC\$_NO_PASSALL
SJC\$_PRIORITY	
SJC\$_RESTART	SJC\$_NO_RESTART
SJC\$_UIC	
SJC\$_USERNAME	
SJC\$_WSDEFAULT	SJC\$_NO_WSDEFAULT
SJC\$_WSEXTENT	SJC\$_NO_WSEXTENT
SJC\$_WSQUOTA	SJC\$_NO_WSQUOTA

The following output item codes may be specified:

SJC\$_ENTRY_NUMBER_OUTPUT
SJC\$_JOB_STATUS_OUTPUT

SJC\$_MERGE_QUEUE

This request requeues all jobs in the queue specified by the item code SJC\$_QUEUE to the queue specified by the item code SJC\$_DESTINATION_QUEUE. The execution of current jobs is unaffected.

The following input item codes must be specified:

SJC\$_QUEUE
SJC\$_DESTINATION_QUEUE

SJC\$_PAUSE_QUEUE

This request pauses execution of current jobs in the specified queue and prevents the starting of jobs in that queue.

The following input item code must be specified:

SJC\$_QUEUE

SJC\$_RESET_QUEUE

This request resets the specified queue by (1) terminating and deleting each executing job that is not restartable, (2) terminating and requeuing each executing job that is restartable, and (3) stopping the queue.

The following input item code must be specified:

SJC\$_QUEUE

SJC\$_START_ACCOUNTING

This request performs two functions. If the SJC\$_ACCOUNTING_TYPES item code is specified, the request enables recording of the specified types of accounting records; if SJC\$_ACCOUNTING_TYPES is not specified, the request starts the accounting manager and opens the system accounting file.

System Service Descriptions

\$QIO

mechanism: **by value**

I/O channel that is assigned to the device to which the request is directed. The **chan** argument is a word value containing the number of the I/O channel; however, \$QIO uses only the low-order word.

func

VMS Usage: **function_code**

type: **word (unsigned)**

access: **read only**

mechanism: **by value**

Device-specific function codes and function modifiers specifying the operation to be performed. The **func** is a longword value containing the function code.

Each device has its own function codes and function modifiers. Refer to the *VAX/VMS I/O Reference Volume* for complete information about the function codes and function modifiers that apply to the particular device to which the I/O operation is to be directed.

iosb

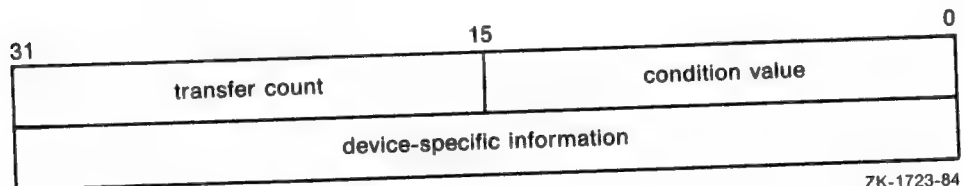
VMS Usage: **io_status_block**

type: **quadword (unsigned)**

access: **write only**

mechanism: **by reference**

I/O status block to receive the final completion status of the I/O operation. The **iosb** is the address of the quadword I/O status block. The following diagram depicts the structure of the I/O status block:



ZK-1723-84

I/O Status Block Fields

condition value

Word-length condition value returned by \$QIO when the I/O operation actually completes.

transfer count

Number of bytes of data actually transferred in the I/O operation. Refer to the *VAX/VMS I/O Reference Volume* for information on how specific devices handle this field of the I/O status block.

device-specific information

The contents of this field vary depending on the specific device and on the specified function code. Again, refer to the *VAX/VMS I/O Reference Volume* for information on how specific devices handle this field of the I/O status block.

System Service Descriptions

\$QIO

When \$QIO begins execution, it clears the quadword I/O status block if the **iosb** argument is specified.

Though this argument is optional, DIGITAL strongly recommends that you specify it for the following reasons.

- If you are using an event flag to signal the completion of the service, you can test the I/O status block for a condition value to be sure that the event flag was not set by an event other than service completion.
- If you are using the \$SYNCH service to synchronize completion of the service, the I/O status block is a required argument for \$SYNCH.
- The condition value returned in R0 and the condition value returned in the I/O status block provide information about different aspects of the call to the \$QIO service. The condition value returned in R0 gives you information about the success or failure of the service call itself; the condition value returned in the I/O status block gives you information about the success or failure of the service operation. Therefore, to accurately assess the success or failure of the call to \$QIO, you must check the condition values returned in both R0 and the I/O status block.

astadr

VMS Usage: **ast_procedure**
type: **procedure entry mask**
access: **call without stack unwinding**
mechanism: **by reference**

AST service routine to be executed when the I/O completes. The **astadr** argument is the address of a longword value that is the entry mask to the AST routine.

The AST routine executes at the access mode of the caller of \$QIO.

astprm

VMS Usage: **user_arg**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

AST parameter to be passed to the AST service routine. The **astprm** argument is a longword value containing the AST parameter.

p1 to p6

VMS Usage: **varying_arg**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Optional device- and function-specific I/O request parameters.

For more information on these parameters see the *VAX/VMS I/O Reference Volume*.

System Service Descriptions

\$SNDJBC

SJC\$_ACCOUNTING_TYPES
SJC\$_NEW_VERSION

SJC\$_START_QUEUE

This request permits the starting of jobs in the specified queue. If the queue was paused, current jobs are resumed.

The following input item code must be specified:

SJC\$_QUEUE

The following input or Boolean item codes may be specified:

SJC\$_ALIGNMENT_MASK	
SJC\$_ALIGNMENT_PAGES	
SJC\$_BASE_PRIORITY	
SJC\$_BATCH	SJC\$_NO_BATCH
SJC\$_CHARACTERISTIC_NAME	SJC\$_NO_CHARACTERISTICS
SJC\$_CHARACTERISTIC_NUMBER	
SJC\$_CPU_DEFAULT	SJC\$_NO_CPU_DEFAULT
SJC\$_CPU_LIMIT	SJC\$_NO_CPU_LIMIT
SJC\$_DEFAULT_FORM_NAME	
SJC\$_DEFAULT_FORM_NUMBER	
SJC\$_DEVICE_NAME	
SJC\$_FILE_BURST	SJC\$_NO_FILE_BURST
SJC\$_FILE_BURST_ONE	
SJC\$_FILE_FLAG	SJC\$_NO_FILE_FLAG
SJC\$_FILE_FLAG_ONE	
SJC\$_FILE_TRAILER	SJC\$_NO_FILE_TRAILER
SJC\$_FILE_TRAILER_ONE	
SJC\$_FORM_NAME	
SJC\$_FORM_NUMBER	
SJC\$_GENERIC_QUEUE	SJC\$_NO_GENERIC_QUEUE
SJC\$_GENERIC_SELECTION	SJC\$_NO_GENERIC_SELECTION
SJC\$_GENERIC_TARGET	
SJC\$_JOB_BURST	SJC\$_NO_JOB_BURST
SJC\$_JOB_FLAG	SJC\$_NO_JOB_FLAG
SJC\$_JOB_LIMIT	
SJC\$_JOB_RESET_MODULES	SJC\$_NO_JOB_RESET_MODULES
SJC\$_JOB_SIZE_MAXIMUM	SJC\$_NO_JOB_SIZE_MAXIMUM
SJC\$_JOB_SIZE_MINIMUM	SJC\$_NO_JOB_SIZE_MINIMUM
SJC\$_JOB_SIZE_SCHEDULING	SJC\$_NO_JOB_SIZE_SCHEDULING
SJC\$_JOB_TRAILER	SJC\$_NO_JOB_TRAILER
SJC\$_LIBRARY_SPECIFICATION	SJC\$_NO_LIBRARY_SPECIFICATION
SJC\$_NEXT_JOB	
SJC\$_OWNER_UIC	

System Service Descriptions

SSNDJBC

SJC\$_PAGINATE	SJC\$_NO_PAGINATE
SJC\$_PROCESSOR	SJC\$_NO_PROCESSOR
SJC\$_PROTECTION	
SJC\$_RECORD_BLOCKING	SJC\$_NO_RECORD_BLOCKING
SJC\$_RELATIVE_PAGE	
SJC\$_RETAIN_ALL_JOBS	SJC\$_NO_RETAIN_JOBS
SJC\$_RETAIN_ERROR_JOBS	
SJC\$_SCSNODE_NAME	
SJC\$_SEARCH_STRING	
SJC\$_SWAP	SJC\$_NO_SWAP
SJC\$_TERMINAL	SJC\$_NO_TERMINAL
SJC\$_TOP_OF_FILE	
SJC\$_WSDEFAULT	SJC\$_NO_WSDEFAULT
SJC\$_WSEXTENT	SJC\$_NO_WSEXTENT
SJC\$_WSQUOTA	SJC\$_NO_WSQUOTA

SJC\$_START_QUEUE_MANAGER

This request starts the queue manager applying file specification defaults from SYS\$SYSTEM:JBCSYSQUE.DAT; it either opens an existing system job queue file or creates a new one. Use of the SJC\$_NEW_VERSION item code forces the creation of a new system job queue file.

SJC\$_BUFFER_COUNT	
SJC\$_EXTEND_QUANTITY	
SJC\$_NEW_VERSION	
SJC\$_QUEUE_FILE_SPECIFICATION	
SJC\$_QUEMAN_RESTART	SJC\$_NO_QUEMAN_RESTART

SJC\$_STOP_ACCOUNTING

This request performs two functions. If the SJC\$_ACCOUNTING_TYPES item code is specified, the request disables recording of the specified types of accounting records. If SJC\$_ACCOUNTING_TYPES is not specified, the request stops the accounting manager and closes the system accounting file.

SJC\$_ACCOUNTING_TYPES

SJC\$_STOP_QUEUE

This request prevents the starting of jobs in the specified queue. The execution of current jobs is unaffected.

The following input item code must be specified:

SJC\$_QUEUE

SJC\$_STOP_QUEUE_MANAGER

This request shuts down the queue manager: it stops each queue that is managed by the requesting node; it aborts each job that is currently executing, requeuing those jobs that are restartable; and closes the system job queue file. No item codes are allowed.

System Service Descriptions

\$SNDJBC

SJC\$_SYNCHRONIZE_JOB

This request waits for completion of a job, then sets the event flag, declares the completion AST, and returns the completion status of the job to the I/O Status Block, providing the **iosb** argument was specified in the call to the \$SNDJBC service.

The following input item code must be specified:

SJC\$_QUEUE

One of the following input item codes must be specified:

SJC\$_ENTRY_NUMBER

SJC\$_JOB_NAME

SJC\$_WRITE_ACCOUNTING

This request writes an accounting record.

The following input item code must be specified:

SJC\$_ACCOUNTING_MESSAGE

nullarg

VMS Usage: **null_arg**

type: **longword (unsigned)**

access: **read only**

mechanism: **by value**

Place-holding argument. This argument is reserved to DIGITAL.

itmlst

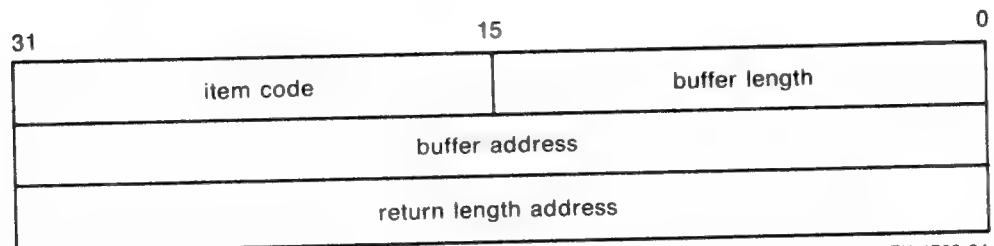
VMS Usage: **item_list_3**

type: **longword (unsigned)**

access: **read only**

mechanism: **by reference**

Item list supplying information to be used in performing the function specified by the **func** argument. The **itmlst** argument is the address of the item list. The item list consists of one or more item descriptors, each of which specifies an item code. The item list is terminated by an item code of 0 or by a longword of 0. The following diagram depicts the structure of a single item descriptor:



ZK-1705-84

\$SNDJBC Item Descriptor Fields

buffer length

A word specifying the length of the buffer; the buffer either supplies information to be used by \$SNDJBC or receives information from \$SNDJBC. The required length of the buffer varies depending on the item code specified and is given in the description of each item code.

System Service Descriptions

\$SNDJBC

item code

A word containing an item code, which identifies the nature of the information that is supplied for use by \$SNDJBC or that is received from \$SNDJBC. Each item code has a symbolic name; these symbolic names are defined by the \$SJCDEF macro and have the format: SJC\$_code.

There are three types of item code:

- Boolean item code. Boolean item codes specify a true or false value: the form SJC\$_code specifies a true value; SJC\$_NO_code specifies a false value. For Boolean item codes, the **buffer length**, **buffer address**, and **return length** fields of the item descriptor must be zero.
- Input value item code. Input value item codes specify an input value to be used by \$SNDJBC. For input value item codes, the **buffer length** and **buffer address** fields of the item descriptor must be nonzero; the **return length** field must be zero. Specific buffer length requirements are given in the description of each item code.
- Output value item code. Output value item codes specify a buffer for information returned by \$SNDJBC. For output value item codes, the **buffer length** and **buffer address** fields of the item descriptor must be nonzero; the **return length** field may be zero or nonzero. Specific buffer length requirements are given in the description of each item code.

Several item codes specify a queue name, form name, or characteristic name. For these item codes, the buffer must specify a string containing from 1 to 31 characters, exclusive of spaces, tabs, and null characters, which are ignored. Allowable characters in the string are the uppercase alphabetic characters, the lowercase alphabetic characters (which are converted to uppercase), the numeric characters, the dollar sign (\$), and the underscore (_).

buffer address

Address of the buffer that specifies or receives the information.

return length address

Address of a word to receive the length in bytes of information returned by \$SNDJBC. If this address is specified as 0, no length is returned.

\$SNDJBC Item Codes

SJC\$_ACCOUNT_NAME

SJC\$_ACCOUNT_NAME is an input value item code. This value specifies the 8-byte account name of the user on behalf of whom the request is made. By default, the account name is taken from the requesting process.

This item code requires CMKRNL privilege.

SJC\$_ACCOUNTING_MESSAGE

SJC\$_ACCOUNTING_MESSAGE is an input value item code. It causes the contents of the buffer to be placed in the "user data" accounting record. The buffer must specify a string of from 1 to 255 characters.

SJC\$_ACCOUNTING_TYPES

SJC\$_ACCOUNTING_TYPES is an input value item code. It enables or disables accounting-record types. When an accounting-record type is enabled, the event designated by that type will be recorded in the accounting record. The buffer must contain a longword bit mask, wherein each bit specifies an accounting-record type. Undefined bits must be zero.

System Service Descriptions

\$SNDJBC

These types have symbolic names, which are defined by the \$SJCDEF macro. The longword bit mask is constructed by performing a logical OR of the symbolic names of each desired accounting-record type. Following is a list of each accounting-record type and the system event to which it corresponds:

Accounting-Record Type	Corresponding System Event
SJC\$V_ACCT_IMAGE	Image terminations
SJC\$V_ACCT_LOGIN_FAILURE	Login failures
SJC\$V_ACCT_MESSAGE	User messages sent
SJC\$V_ACCT_PRINT	Print job terminations
SJC\$V_ACCT_PROCESS	Process terminations

The following accounting-record types, when enabled, provide additional information about image and process termination; specifically, they describe the type of image or process that has terminated.

Accounting-Record Type	Type of Image or Process
SJC\$V_ACCT_BATCH	Batch process
SJC\$V_ACCT_DETACHED	Detached process
SJC\$V_ACCT_INTERACTIVE	Interactive process
SJC\$V_ACCT_NETWORK	Network process
SJC\$V_ACCT_SUBPROCESS	Subprocess

SJC\$_AFTER_TIME

SJC\$_NO_AFTER_TIME

SJC\$_AFTER_TIME is an input value item code. It specifies that the job can execute only if the system time is greater than or equal to the quadword time value contained in the buffer. The buffer must contain either an absolute time value or a delta time value; \$SNDJBC converts delta time values to absolute time values by adding the current system time.

SJC\$_NO_AFTER_TIME is a Boolean item code. It cancels the effect of a SJC\$_AFTER_TIME item code previously specified for the job; the job can execute immediately. It is the default.

SJC\$_ALIGNMENT_MASK

SJC\$_ALIGNMENT_MASK is a Boolean item code. It is meaningful only for output execution queues and only when the SJC\$_ALIGNMENT_PAGES item code is also specified. SJC\$_ALIGNMENT_MASK causes the data printed on form alignment pages to be masked: all alphabetic characters are replaced with the letter "X" and all numeric characters with the number "9".

SJC\$_ALIGNMENT_PAGES

SJC\$_ALIGNMENT_PAGES is an input value item code. It is meaningful only for output execution queues. It specifies that the queue be placed in form-alignment state and that a number of alignment pages be printed. The buffer must contain a longword value in the range 1 to 20; this value specifies how many alignment pages are to be printed.

SJC\$_BASE_PRIORITY

SJC\$_BASE_PRIORITY is an input value item code. It is meaningful only for execution queues. It specifies the base priority of batch processes initiated from a batch execution queue or of a symbiont process connected to an output

System Service Descriptions

\$SNDJBC

execution queue. A symbiont process can control several queues; however, the base priority of the symbiont process is established by the first queue to which it is connected. The buffer must contain a longword value in the range 0 to 15; this value specifies the base priority.

By default, the base priority is the value of the SYSGEN parameter DEFPRI. If the value of DEFPRI is 0, the default base priority is the base priority of the requesting process.

SJC\$_BATCH

SJC\$_NO_BATCH

SJC\$_BATCH is a Boolean item code. It specifies that the queue is a batch execution queue or a generic batch queue, and thus can process only batch jobs.

SJC\$_NO_BATCH is a Boolean item code. It specifies that the queue is not a batch queue but rather an output execution or generic output queue, and thus can process only print jobs. It is the default.

SJC\$_BUFFER_COUNT

SJC\$_BUFFER_COUNT is an input value item code. It specifies the number of buffers that the job controller should allocate to its local buffer cache for performing I/O operations to the system job queue file. The buffer must contain a longword integer value in the range 1 through 127 or 0; this value specifies the number of buffers the job controller allocates to its local buffer cache. If zero is specified, a default value of 50 is used.

SJC\$_CHARACTERISTIC_NAME

SJC\$_CHARACTERISTIC_NUMBER

SJC\$_NO_CHARACTERISTICS

SJC\$_CHARACTERISTIC_NAME and SJC\$_CHARACTERISTIC_NUMBER are both input value item codes. Both specify characteristics for jobs or queues, and they may be used interchangeably. The characteristics are user-defined.

The SJC\$_DEFINE_CHARACTERISTIC and SJC\$_DELETE_CHARACTERISTIC function codes include and delete, respectively, a specified characteristic from the system job queue file. A job cannot execute on an execution queue unless the queue possesses all the characteristics possessed by the job; the queue may possess additional characteristics and the job will still execute.

The SJC\$_CHARACTERISTIC_NAME and SJC\$_CHARACTERISTIC_NUMBER item codes may appear as many times as necessary in a single call to \$SNDJBC; the set of characteristics so defined in the call completely replaces the set of characteristics defined by a prior call. The SJC\$_NO_CHARACTERISTICS item code cancels all defined characteristics for the job or queue. By default, a queue or job has no defined characteristics.

The string may contain uppercase or lowercase characters (lowercase are converted to uppercase), numeric characters, dollar signs, and underscores. If the string is a logical name, SYS\$SNDJBC translates it iteratively until the equivalence string is found or the number of translations allowed by the system has been performed. The maximum length of the final character string is 31 characters; spaces, tabs, and null characters are ignored.

For SJC\$_CHARACTERISTIC_NUMBER, the buffer must contain a longword value in the range 0 to 127. This number identifies a characteristic.

SJC\$_NO_CHARACTERISTICS is a Boolean item code.

SJCS_CHECKPOINT_DATA

SJCS_NO_CHECKPOINT_DATA

SJCS_CHECKPOINT_DATA is an input value item code. It specifies the value of the DCL symbol **BATCH\$RESTART** for a batch job that is restarted. The buffer must contain a string no longer than 255 characters; this string is the value of the symbol **BATCH\$RESTART**.

SJCS_NO_CHECKPOINT_DATA is a Boolean item code. It cancels a previous specification of the **BATCH\$RESTART** symbol; it also cancels a checkpoint in a print job so that the entire job is reprinted. By default, the **BATCH\$RESTART** symbol is undefined.

SJCS_CLI

SJCS_NO_CLI

SJCS_CLI is an input value item code. It is meaningful only for batch jobs. It specifies that the command language interpreter to be executed is **SYS\$SYSTEM:name.EXE**, where "name" is a valid RMS file name. The buffer must specify a name string of from 1 to 39 characters.

SJCS_NO_CLI is a Boolean item code. It specifies that the command language interpreter to be executed is the one specified in the user's authorization file. It is the default.

SJCS_CPU_DEFAULT

SJCS_NO_CPU_DEFAULT

SJCS_CPU_DEFAULT is an input value item code. It is meaningful only for batch execution queues. It specifies the default CPU time limit in 10 millisecond units. The buffer contains this longword value. The value 0 specifies unlimited CPU time.

SJCS_NO_CPU_DEFAULT is a Boolean item code. It is meaningful only for batch execution queues. It specifies that no default CPU time limit is to apply to the job. It is the default.

A CPU time limit for the process is included in each user record in the system user authorization file (UAF). You can also specify any or all of the following: a CPU time limit for individual jobs, a default CPU time limit for all jobs in a given queue, and a maximum CPU time limit for all jobs in a given queue. Table SYS-6 shows the action taken when a value is specified for **SJCS_CPU_DEFAULT**.

Table SYS-6 CPU Time Limit Decision Table

CPU Time Limit Specified for Job?	Default CPU Time Limit Specified for Queue?	Maximum CPU Time Specified for Queue?	Action Taken
No	No	No	Use UAF value
Yes	No	No	Use smaller of job's limit and UAF value
Yes	Yes	No	Use smaller of job's limit and UAF value
Yes	No	Yes	Use smaller of job's limit and maximum

System Service Descriptions

SSNDJBC

Table SYS-6 (Cont.) CPU Time Limit Decision Table

CPU Time Limit Specified for Job?	Default CPU Time Limit Specified for Queue?	Maximum CPU Time Specified for Queue?	Action Taken
Yes	Yes	Yes	Use smaller of job's limit and maximum
No	Yes	Yes	Use smaller of queue's default and maximum
No	No	Yes	Use maximum
No	Yes	No	Use smaller of UAF value and queue's default

SJC\$_CPU_LIMIT

SJC\$_NO_CPU_LIMIT

SJC\$_CPU_LIMIT is an input value item code. It is meaningful only for batch execution queues and batch jobs. It specifies the maximum CPU time limit in 10 millisecond units. The buffer must contain this longword value. The value 0 specifies unlimited CPU time.

SJC\$_NO_CPU_LIMIT is a Boolean item code. It is meaningful only for batch execution queues and batch jobs. It specifies that no maximum CPU time limit is to apply to the job. It is the default.

Refer to the description of the SJC\$_CPU_DEFAULT item code and to Table SYS-6 for information on the action taken when a value is specified for SJC\$_CPU_LIMIT.

SJC\$_CREATE_START

SJC\$_CREATE_START is a Boolean item code. It specifies that a queue be started after it is created. By default, a queue remains stopped after it is created.

SJC\$_DEFAULT_FORM_NAME

SJC\$_DEFAULT_FORM_NUMBER

SJC\$_DEFAULT_FORM_NAME and SJC\$_DEFAULT_FORM_NUMBER are input value item codes. They specify the default form for a specific output queue by name and by number, respectively.

When you specify a default form for an output queue, the queue uses the queue-specific default form, rather than the systemwide default form, to process any job that does not explicitly specify a form.

For SJC\$_DEFAULT_FORM_NAME, the buffer must specify a form name. For SJC\$_DEFAULT_FORM_NUMBER, the buffer must specify a longword value. You should use only one of these item codes to identify a default form for the queue.

SJC\$_DELETE_FILE

SJC\$_NO_DELETE_FILE

SJC\$_DELETE_FILE is a Boolean item code. It specifies that a file be deleted after the job has completed. This item code cannot be specified with the

System Service Descriptions

\$SNDJBC

SJC\$_ALTER_JOB function code, which alters the parameters for an already existing job; a file deletion request can only be made when a job is first submitted to the queue.

SJC\$_NO_DELETE_FILE is a Boolean item code. It specifies that a file not be deleted after execution of the job. It is the default. This item code can be specified with the **SJC\$_ALTER_JOB** function code; this makes it possible to cancel a file deletion request that was made when the job was first submitted to the queue.

SJC\$_DESTINATION_QUEUE

SJC\$_DESTINATION_QUEUE is an input value item code. When the **SJC\$_ASSIGN_QUEUE** function code is specified, **SJC\$_DESTINATION_QUEUE** specifies the name of the execution queue to which the logical queue is assigned. When the **SJC\$_ABORT_JOB**, **SJC\$_ALTER_JOB**, or **SJC\$_MERGE_QUEUE** function codes are specified, **SJC\$_DESTINATION_QUEUE** specifies the name of the queue into which jobs are placed. By default, jobs remain in the original queue.

The string may contain uppercase or lowercase characters (lowercase are converted to uppercase), numeric characters, dollar signs, and underscores. If the string is a logical name, **SYS\$SNDJBC** translates it iteratively until the equivalence string is found or the number of translations allowed by the system has been performed. The maximum length of the final character string is 31 characters; spaces, tabs, and null characters are ignored.

SJC\$_DEVICE_NAME

SJC\$_DEVICE_NAME is an input value item code. It is meaningful only for execution queues. It specifies the node and/or device on which the specified execution queue is located. For batch queues, only the node name can be specified. The buffer must specify a string of from 1 to 31 characters.

SJC\$_DOUBLE_SPACE

SJC\$_NO_DOUBLE_SPACE

SJC\$_DOUBLE_SPACE is a Boolean item code. It is meaningful only for output execution queues. It specifies that the symbiont should print the file with double spacing.

SJC\$_NO_DOUBLE_SPACE is a Boolean item code. It specifies that the symbiont should print the file with single spacing. It is the default.

SJC\$_ENTRY_NUMBER

SJC\$_ENTRY_NUMBER is an input value item code. It specifies the entry number of the job on which to perform the function. The buffer must contain this entry number.

SJC\$_ENTRY_NUMBER_OUTPUT

SJC\$_ENTRY_NUMBER_OUTPUT is an output value item code. The buffer must specify a longword into which **\$SNDJBC** will write the entry number of a created job.

SJC\$_EXTEND_QUANTITY

SJC\$_EXTEND_QUANTITY is an input value item code. It specifies the system job queue file extension size in blocks. This extension size is used when the queue file is extended. This value is also used to establish an initial allocation size for the queue file when it is created. The buffer must contain a longword integer value in the range 10 through 65,535 or 0. This value specifies the number of blocks by which the queue should be extended. The

System Service Descriptions

SSNDJBC

default value is 100 blocks. If a value of 0 is specified, the default size is used.

SJC\$_FILE_BURST **SJC\$_FILE_BURST_ONE** **SJC\$_NO_FILE_BURST**

SJC\$_FILE_BURST is a Boolean item code. It is meaningful only for output execution queues. It specifies that a burst page is to be printed preceding a file. If SJC\$_FILE_BURST is specified for a job, it specifies the default for all files in the job; if specified for an output execution queue, it specifies the default for all jobs executed from that queue.

SJC\$_FILE_BURST_ONE is a Boolean item code. It is meaningful only for output execution queues. It specifies that a burst page is to be printed preceding a file. If SJC\$_FILE_BURST_ONE is specified for a job, it specifies that a burst page is to be printed preceding only the first copy of the first file in the job; if specified for an output execution queue, it specifies this behavior as the default for all jobs executed from that queue.

SJC\$_NO_FILE_BURST is a Boolean item code. It is meaningful only for output execution queues. It specifies that no burst page be printed. It is the default.

SJC\$_FILE_COPIES

SJC\$_FILE_COPIES is an input value item code. It is meaningful only for output execution queues. It specifies the number of times a file is repeated. By default, a file is repeated once. The buffer must specify a longword value from 1 to 255; this value is the repeat count.

SJC\$_FILE_FLAG **SJC\$_FILE_FLAG_ONE** **SJC\$_NO_FILE_FLAG**

SJC\$_FILE_FLAG is a Boolean item code. It is meaningful only for output execution queues. It specifies that a flag page is to be printed preceding a file. If SJC\$_FILE_FLAG is specified for a job, it specifies the default for all files in the job; if specified for an output execution queue, it specifies the default for all jobs executed from that queue.

SJC\$_FILE_FLAG_ONE is a Boolean item code. It is meaningful only for output execution queues. It specifies that a flag page is to be printed preceding a file. If SJC\$_FILE_FLAG_ONE is specified for a job, it specifies that a flag page is to be printed preceding only the first copy of the first file in the job; if specified for an output execution queue, it specifies this behavior as the default for all jobs executed from that queue.

SJC\$_NO_FILE_FLAG is a Boolean item code. It is meaningful only for output execution queues. It specifies that no flag page be printed. It is the default.

SJC\$_FILE_IDENTIFICATION

SJC\$_FILE_IDENTIFICATION is an input value item code. It specifies the file to be processed. The buffer contains a 28-byte value that identifies the file to be processed. This value specifies (in order) the following three file-identification fields in the RMS NAM block: the 16-byte NAM\$_T_DVI field, the 6-byte NAM\$_W_FID field, and the 6-byte NAM\$_W_DID field. These fields occur in the NAM block consecutively and in the same order as they are listed above.

If SJC\$_FILE_IDENTIFICATION is specified, the SJC\$_FILE_SPECIFICATION item code must be omitted.

System Service Descriptions

\$SNDJBC

SJC\$_FILE_SETUP_MODULES **SJC\$_NO_FILE_SETUP_MODULES**

SJC\$_FILE_SETUP_MODULES is an input value item code. It is meaningful only for output execution queues. It specifies that a list of text modules be extracted from the device control library and copied to the printer before a file is printed. The buffer must contain a list of text module names, with a comma separating each name.

SJC\$_NO_FILE_SETUP_MODULES is a Boolean item code. It is meaningful only for output execution queues. It specifies that no text modules be copied before printing a file. It is the default.

SJC\$_FILE_SPECIFICATION

SJC\$_FILE_SPECIFICATION is an input value item code. It identifies the file to be processed. The buffer must contain the file specification of the file to be processed. The \$SNDJBC service converts the file specification to the corresponding file identification and proceeds as if the SJC\$_FILE_IDENTIFICATION item code had been specified. If SJC\$_FILE_SPECIFICATION is specified, the SJC\$_FILE_IDENTIFICATION item code must be omitted.

SJC\$_FILE_TRAILER **SJC\$_FILE_TRAILER_ONE** **SJC\$_NO_FILE_TRAILER**

SJC\$_FILE_TRAILER is a Boolean item code. It is meaningful only for output execution queues. It specifies that a trailer page is to be printed following a file. If SJC\$_FILE_TRAILER is specified for a job, it specifies the default for all files in the job; if specified for an output execution queue, it specifies the default for all jobs executed on that queue.

SJC\$_FILE_TRAILER_ONE is a Boolean item code. It is meaningful only for output execution queues. It specifies that a trailer page is to be printed following a file. If SJC\$_FILE_TRAILER_ONE is specified for a job, it specifies that a trailer page is to be printed following only the last copy of the last file in the job; if specified for an output execution queue, it specifies this behavior as the default for all jobs executed on that queue.

SJC\$_NO_FILE_TRAILER is a Boolean item code. It is meaningful only for output execution queues. It specifies that no trailer page be printed. It is the default.

SJC\$_FIRST_PAGE **SJC\$_NO_FIRST_PAGE**

SJC\$_FIRST_PAGE is an input value item code. It is meaningful only for output execution queues. It specifies the page number at which printing should begin. The buffer must contain a nonzero longword value specifying this page number.

SJC\$_NO_FIRST_PAGE is a Boolean item code. It is meaningful only for output execution queues. It specifies that printing should begin with the first page. It is the default.

SJC\$_FORM_DESCRIPTION

SJC\$_FORM_DESCRIPTION is an input value item code. It specifies text that describes the form to users and operators. By default, the form name is used. The buffer must specify a string of no more than 255 characters.

System Service Descriptions

\$SNDJBC

SJC\$_FORM_LENGTH

SJC\$_FORM_LENGTH is an input value item code. It is meaningful only for output execution queues. It specifies the physical length of the form in lines. The buffer must contain a nonzero longword integer value. By default, the form length is 66 lines.

SJC\$_FORM_MARGIN_BOTTOM

SJC\$_FORM_MARGIN_BOTTOM is an input value item code. It specifies the bottom margin of the form in lines. By default, the bottom margin is 6 lines.

SJC\$_FORM_MARGIN_LEFT

SJC\$_FORM_MARGIN_LEFT is an input value item code. It specifies the left margin of the form in characters. By default, the left margin is 0. The buffer must specify a longword value.

SJC\$_FORM_MARGIN_RIGHT

SJC\$_FORM_MARGIN_RIGHT is an input value item code. It specifies the right margin of the form in characters. By default, the right margin is 0. The buffer must specify a longword value.

SJC\$_FORM_MARGIN_TOP

SJC\$_FORM_MARGIN_TOP is an input value item code. It specifies the top margin of the form in lines. By default, the top margin is 0.

SJC\$_FORM_NAME

SJC\$_FORM_NUMBER

SJC\$_FORM_NAME and SJC\$_FORM_NUMBER are input value item codes. They specify a mounted form for the queue by name and by number, respectively. For SJC\$_FORM_NAME, the buffer must specify a form name. For SJC\$_FORM_NUMBER, the buffer must specify a longword value. You should use only one of these two item codes to identify a mounted form for the queue.

The SJC\$_DEFINE_FORM and SJC\$_DELETE_FORM function codes include and delete, respectively, a specified form name and number from the system job queue file. The mounted form indicates the stock type of the output queue. A job cannot execute on an output queue unless the stock type of the form specified (by name or number) for the job is the same as the stock type of the mounted form specified for the queue. For more information on how the stock type of a form affects job processing, see Chapter 9 of the *VAX/VMS System Manager's Reference Manual*.

The string may contain uppercase or lowercase characters (lowercase are converted to uppercase), numeric characters, dollar signs, and underscores. If the string is a logical name, SYS\$SNDJBC translates it iteratively until the equivalence string is found or the number of translations allowed by the system has been performed. The maximum length of the final character string is 31 characters; spaces, tabs, and null characters are ignored.

SJC\$_FORM_SETUP_MODULES

SJC\$_NO_FORM_SETUP_MODULES

SJC\$_FORM_SETUP_MODULES is an input value item code. It is meaningful only for output execution queues. The buffer must specify one or more text module names, with a comma separating each name. This item code specifies that these modules be extracted from the device control library and copied to the printer before each file that is printed on the form.

System Service Descriptions

\$SNDJBC

SJC\$_NO_FORM_SETUP_MODULES is a Boolean item code. It specifies that no device control modules be copied. It is the default.

SJC\$_FORM_SHEET_FEED **SJC\$_NO_FORM_SHEET_FEED**

SJC\$_FORM_SHEET_FEED is a Boolean item code. It specifies that the symbiont should pause at the end of each physical page so that a new form may be inserted.

SJC\$_NO_FORM_SHEET_FEED is a Boolean item code. It specifies that the output symbiont should not pause at the end of every physical page. It is the default.

SJC\$_FORM_STOCK

SJC\$_FORM_STOCK is an input value item code. It specifies a name for the paper stock. The buffer must contain a string of 1 to 31 characters. By default, the name of the paper stock is the form name.

SJC\$_FORM_TRUNCATE **SJC\$_NO_FORM_TRUNCATE**

SJC\$_FORM_TRUNCATE is a Boolean item code. It specifies that the symbiont should truncate lines that extend beyond the right margin. Specifying **SJC\$_FORM_TRUNCATE** cancels **SJC\$_FORM_WRAP**. It is the default.

SJC\$_NO_FORM_TRUNCATE is a Boolean item code. It specifies that the output symbiont should not truncate lines that extend beyond the right margin.

SJC\$_FORM_WIDTH

SJC\$_FORM_WIDTH is an input value item code. It specifies the physical width of the form in characters. The buffer must contain a nonzero longword integer. By default, the form width is 132 characters.

SJC\$_FORM_WRAP **SJC\$_NO_FORM_WRAP**

SJC\$_FORM_WRAP is a Boolean item code. It specifies that the symbiont should wrap lines that extend beyond the right margin. Specifying **SJC\$_FORM_WRAP** cancels **SJC\$_FORM_TRUNCATE**.

SJC\$_NO_FORM_WRAP is a Boolean item code. It specifies that the output symbiont should not wrap lines. It is the default.

SJC\$_GENERIC_QUEUE **SJC\$_NO_GENERIC_QUEUE**

SJC\$_GENERIC_QUEUE is a Boolean item code. It specifies that a queue is a generic queue.

SJC\$_NO_GENERIC_QUEUE is a Boolean item code. It specifies that a queue is not a generic queue. It is the default. By default, a queue is an execution queue; see the Description section for a full discussion of the types of queue.

SJC\$_GENERIC_SELECTION **SJC\$_NO_GENERIC_SELECTION**

SJC\$_GENERIC_SELECTION is a Boolean item code. It specifies that an execution queue can accept work from a generic queue. It is the default. It is meaningful only for execution queues.

System Service Descriptions

\$SNDJBC

SJC\$_NO_GENERIC_SELECTION is a Boolean item code. It specifies that an execution queue cannot accept work from a generic queue.

SJC\$_GENERIC_TARGET

SJC\$_GENERIC_TARGET is an input value item code. The buffer must specify a queue name. This queue name identifies an execution queue that can accept work from a generic queue. This item code is meaningful only for generic queues.

This item code can appear up to 124 times in a single call to \$SNDJBC. The set of queues defined in a single call completely replaces the set defined by a prior call.

The string may contain uppercase or lowercase characters (lowercase are converted to uppercase), numeric characters, dollar signs, and underscores. If the string is a logical name, SYS\$SNDJBC translates it iteratively until the equivalence string is found or the number of translations allowed by the system has been performed. The maximum length of the final character string is 31 characters; spaces, tabs, and null characters are ignored.

SJC\$_HOLD

SJC\$_NO_HOLD

SJC\$_HOLD is a Boolean item code. It specifies that a job cannot execute and must enter a holding status.

SJC\$_NO_HOLD is a Boolean item code. It specifies that a job can execute immediately. It makes the following types of job eligible for execution: (1) a job that is holding because it was specified with the **SJC\$_HOLD** item code, (2) a job that is holding at the request of a symbiont, and (3) a job that was retained after execution. It is the default. **SJC\$_NO_HOLD** does not release a job that is holding until a specified time because it was specified with the **SJC\$_AFTER_TIME** item code.

SJC\$_JOB_BURST

SJC\$_NO_JOB_BURST

SJC\$_JOB_BURST is a Boolean item code. It specifies that a burst page is to be printed preceding each job. It is meaningful only for output execution queues.

SJC\$_NO_JOB_BURST is a Boolean item code. It specifies that a burst page is not to be printed preceding each job. It is meaningful only for output execution queues. It is the default.

SJC\$_JOB_COPIES

SJC\$_JOB_COPIES is an input value item code. It specifies the number of times that the entire print job is to be repeated. The buffer must contain this nonzero longword integer value. By default, the print job is repeated once.

SJC\$_JOB_FLAG

SJC\$_NO_JOB_FLAG

SJC\$_JOB_FLAG is a Boolean item code. It specifies that a flag page is to be printed preceding each job. It is meaningful only for output execution queues.

SJC\$_NO_JOB_FLAG is a Boolean item code. It specifies that a flag page is not to be printed preceding each job. It is meaningful only for output execution queues. It is the default.

System Service Descriptions

\$SNDJBC

SJC\$_JOB_LIMIT

SJC\$_JOB_LIMIT is an input value item code. It specifies the maximum number of jobs that can execute simultaneously on a queue. The buffer must contain a longword value in the range 1 to 255. It is meaningful only for batch execution queues. By default, the job limit is 1.

SJC\$_JOB_NAME

SJC\$_JOB_NAME is an input value item code. It specifies the name of a job. The buffer must specify a string of from 1 to 39 characters.

For function codes SJC\$_ENTER_FILE, SJC\$_CREATE_JOB, and SJC\$_ALTER_JOB, SJC\$_JOB_NAME specifies the identifying name of the job. By default, the name used is the name of the first file in the job.

For function code SJC\$_SYNCHRONIZE_JOB, SJC\$_JOB_NAME specifies the name of the job on which to operate. The job name is implicitly qualified by username.

SJC\$_JOB_RESET_MODULES

SJC\$_NO_JOB_RESET_MODULES

SJC\$_JOB_RESET_MODULES is an input value item code. It is meaningful only for output execution queues. The buffer must specify the names of one or more text modules, with a comma separating each name. This item code specifies that these modules are to be extracted from the device control library and copied to the printer between each print job.

SJC\$_NO_JOB_RESET_MODULES is a Boolean item code. It specifies that no text modules be copied to the printer. It is the default.

SJC\$_JOB_SIZE_MAXIMUM

SJC\$_NO_JOB_SIZE_MAXIMUM

SJC\$_JOB_SIZE_MAXIMUM is an input value item code. It is meaningful only for output execution queues. It specifies that a print job can execute only if its total size in blocks is less than or equal to the specified value. The buffer specifies this nonzero longword value.

SJC\$_NO_JOB_SIZE_MAXIMUM is a Boolean item code. It specifies that a print job can execute immediately regardless of its size. It is the default.

SJC\$_JOB_SIZE_MINIMUM

SJC\$_NO_JOB_SIZE_MINIMUM

SJC\$_JOB_SIZE_MINIMUM is an input value item code. It is meaningful only for output execution queues. It specifies that a print job can execute only if its total size in blocks is greater than or equal to the specified value. The buffer specifies this nonzero longword value.

SJC\$_NO_JOB_SIZE_MINIMUM is a Boolean item code. It specifies that a print job can execute immediately regardless of its size. It is the default.

SJC\$_JOB_SIZE_SCHEDULING

SJC\$_NO_JOB_SIZE_SCHEDULING

SJC\$_JOB_SIZE_SCHEDULING is a Boolean item code. It specifies that print jobs entered in an output execution queue be scheduled according to size, with the smallest job of a given priority processed first. It is the default.

SJC\$_NO_JOB_SIZE_SCHEDULING is a Boolean item code. It specifies that print jobs of a given priority not be scheduled according to size.

Changing the value of this item code for a queue while print jobs are pending on any queue will give unpredictable results.

System Service Descriptions

SSNDJBC

SJC\$_JOB_STATUS_OUTPUT

SJC\$_JOB_STATUS_OUTPUT is an output value item code. When specified, SSNDJBC returns as a character string, a textual message describing the status of a submitted job. Since the message can include up to 255 characters, the buffer length field of the item descriptor should specify 255 (bytes).

SJC\$_JOB_TRAILER

SJC\$_NO_JOB_TRAILER

SJC\$_JOB_TRAILER is a Boolean item code. It is meaningful only for output execution queues. It specifies that a trailer page is to be printed following each job.

SJC\$_NO_JOB_TRAILER is a Boolean item code. It is meaningful only for output execution queues. It specifies that a trailer page is not to be printed following each job. It is the default.

SJC\$_LAST_PAGE

SJC\$_NO_LAST_PAGE

SJC\$_LAST_PAGE is an input value item code. It is meaningful only for output execution queues. It specifies the page number at which printing should end. The buffer specifies this nonzero longword value.

SJC\$_NO_LAST_PAGE is a Boolean item code. It specifies that printing should end after the last page. It is the default.

SJC\$_LIBRARY_SPECIFICATION

SJC\$_NO_LIBRARY_SPECIFICATION

SJC\$_LIBRARY_SPECIFICATION is an input value item code. It is meaningful only for output execution queues. It specifies that the device control library is SYS\$LIBRARY:name.TLB, where "name" is a valid RMS file name. The buffer must specify the file name.

SJC\$_NO_LIBRARY_SPECIFICATION is a Boolean item code. It specifies that the device control library is SYS\$LIBRARY:SYSDEVCTL.TLB. It is the default.

SJC\$_LOG_DELETE

SJC\$_NO_LOG_DELETE

SJC\$_LOG_DELETE is a Boolean item code. It specifies that the log file produced for a batch job is to be deleted. It is meaningful only for batch jobs. It is the default.

SJC\$_NO_LOG_DELETE is a Boolean item code. It specifies that the log file produced for a batch job is not to be deleted.

SJC\$_LOG_QUEUE

SJC\$_LOG_QUEUE is an input value item code. It is meaningful only for batch jobs. It specifies the queue into which the log file produced for the batch job is entered for printing. The buffer must specify the name of the queue. By default, the log file is entered in queue SYS\$PRINT.

The string may contain uppercase or lowercase characters (lowercase are converted to uppercase), numeric characters, dollar signs, and underscores. If the string is a logical name, SYS\$SSNDJBC translates it iteratively until the equivalence string is found or the number of translations allowed by the system has been performed. The maximum length of the final character string is 31 characters; spaces, tabs, and null characters are ignored.

System Service Descriptions

\$SNDJBC

SJCS_LOG_SPECIFICATION

SJCS_NO_LOG_SPECIFICATION

SJCS_LOG_SPECIFICATION is an input value item code. It is meaningful only for batch jobs. It specifies the file specification of the log file produced for a batch job. The buffer must contain this RMS file specification. Omitted fields in the file specification are supplied from the default file specification **SYS\$LOGIN:name.LOG**, where "name" is the job name. By default a log file is produced using this default file specification to generate the log file specification.

SJCS_NO_LOG_SPECIFICATION is a Boolean item code. It specifies that no log file be produced for the batch job.

SJCS_LOG_SPOOL

SJCS_NO_LOG_SPOOL

SJCS_LOG_SPOOL is a Boolean item code. It specifies that the log file produced for a batch job is to be printed. It is meaningful only for batch jobs. It is the default.

SJCS_NO_LOG_SPOOL is a Boolean item code. It specifies that the log file for a batch job is not to be printed.

SJCS_LOWERCASE

SJCS_NO_LOWERCASE

SJCS_LOWERCASE is a Boolean item code. It specifies that a job can execute only on a device that has the LOWERCASE device-dependent characteristic. It is meaningful only for output execution queues.

SJCS_NO_LOWERCASE is a Boolean item code. It specifies that a job can execute whether or not the output device has the LOWERCASE device-dependent characteristic. It is the default.

SJCS_NEW_VERSION

SJCS_NEW_VERSION is a Boolean item code. It specifies that a new version of the system job queue file is to be created, whether or not the file already exists. By default, the system job queue file is created only if it does not already exist.

SJCS_NEXT_JOB

SJCS_NEXT_JOB is a Boolean item code. It is meaningful only for output execution queues. It specifies that the current job be aborted and that printing be resumed with the next job.

SJCS_NOTE

SJCS_NO_NOTE

SJCS_NOTE is an input value item code. It is meaningful only for output execution queues. It specifies a string to be printed on the job flag and file flag pages. The buffer must specify this string.

SJCS_NO_NOTE is a Boolean item code. It specifies that no string is to be printed on the job flag and file flag pages. It is the default.

SJCS_NOTIFY

SJCS_NO_NOTIFY

SJCS_NOTIFY is a Boolean item code. It specifies that a message be broadcast, at the time of job completion, to each logged-in terminal, on behalf of the user who submitted the job.

SJCS_NO_NOTIFY is a Boolean item code. It specifies that no message be broadcast at the time of job completion. It is the default.

System Service Descriptions

\$SNDJBC

SJCS_OPERATOR_REQUEST

SJCS_NO_OPERATOR_REQUEST

SJCS_OPERATOR_REQUEST is an input value item code. It is meaningful only for output execution queues. The buffer must contain a text string. This item code specifies that, when a job begins execution, the execution queue is to be placed in the paused state and the specified text string is to be included in a message to the queue operator requesting service.

SJCS_NO_OPERATOR_REQUEST is a Boolean item code. It specifies that the output execution queue is not to be paused and that no message is to be sent to the queue operator. It is the default.

SJCS_OWNER_UIC

SJCS_OWNER_UIC is an input value item code. It specifies the owner UIC of a queue. The buffer must specify the longword UIC. By default, the owner UIC is [1,4].

SJCS_PAGE_HEADER

SJCS_NO_PAGE_HEADER

SJCS_PAGE_HEADER is a Boolean item code. It is meaningful only for output execution queues. It specifies that a page heading is to be printed on each page of output.

SJCS_NO_PAGE_HEADER is a Boolean item code. It specifies that no page heading is to be printed. It is the default.

SJCS_PAGE_SETUP_MODULES

SJCS_NO_PAGE_SETUP_MODULES

SJCS_PAGE_SETUP_MODULES is an input value item code. It is meaningful only for output execution queues. The buffer must specify one or more text module names, with a comma separating each name. This item code specifies that these modules are to be extracted from the device control library and copied to the printer before each page is printed.

SJCS_NO_PAGE_SETUP_MODULES is a Boolean item code. It specifies that no device control modules be copied. It is the default.

SJCS_PAGINATE

SJCS_NO_PAGINATE

SJCS_PAGINATE is a Boolean item code. It is meaningful only for output execution queues. It specifies that the symbiont should paginate the output by inserting a form feed whenever output reaches the bottom margin of the form. It is the default.

SJCS_NO_PAGINATE is a Boolean item code. It specifies that the symbiont should not paginate the output.

SJCS_PARAMETER_1 through SJCS_PARAMETER_8

SJCS_NO_PARAMETERS

SJCS_PARAMETER_1 through SJCS_PARAMETER_8 are input value item codes; there are eight, the last digit of the item code being a number from 1 through 8. For each item code specified, the buffer must specify a string of no more than 255 characters. For batch jobs, the string becomes the value of the DCL symbol P1 through P8, respectively, within the outermost command procedure.

For print jobs, the system makes the string available to the symbiont, though the standard VAX/VMS print symbiont does not use this information. By default, each of the eight parameters specifies a null string.

System Service Descriptions

\$SNDJBC

For function code `SJC$_ALTER_JOB`, if any `SJC$_PARAMETER` item is specified, the value of each unspecified item is the null string.

`SJC$_NO_PARAMETERS` is a Boolean item code. It specifies that none of the `SJC$_PARAMETER` items are to be passed in the batch or print job. It is the default.

SJC\$_PASSALL **SJC\$_NO_PASSALL**

`SJC$_PASSALL` is a Boolean item code. It is meaningful only for output execution queues. It specifies that the symbiont is to print the file in PASSALL mode.

`SJC$_NO_PASSALL` is a Boolean item code. It specifies that the symbiont is not to print the file in PASSALL mode. It is the default.

SJC\$_PRIORITY

`SJC$_PRIORITY` is an input value item code. The buffer must specify a longword value in the range 0 through 255. This value specifies the scheduling priority of the job in a queue relative to the scheduling priority of other jobs in the same queue.

By default, the scheduling priority of the job is the value of the `SYSGEN` parameter `DEFQUEPRI`. If the value of `DEFQUEPRI` is 0, the default scheduling priority is the base priority of the requesting process.

If you specify a value for `SJC$_PRIORITY` that is greater than the `SYSGEN` parameter `MAXQUEPRI` and you do not have either `ALTPRI` or `OPER` privilege, the system uses the value of `MAXQUEPRI`. If you have either `ALTPRI` or `OPER` privilege, the system uses any value you specify for `SJC$_PRIORITY`, even if it is included in the range between `MAXQUEPRI + 1` and 255.

SJC\$_PROCESSOR **SJC\$_NO_PROCESSOR**

`SJC$_PROCESSOR` is an input value item code. The buffer must specify a valid RMS file name.

When specified for an output execution queue, `SJC$_PROCESSOR` specifies that the symbiont image to be executed is `SYS$SYSTEM:name.EXE`, where "name" is the RMS file name contained in the buffer.

When specified for a generic output queue, `SJC$_PROCESSOR` specifies that the generic queue can place jobs only in server queues that are executing the symbiont image `SYS$SYSTEM:name.EXE`, where "name" is the RMS file name contained in the buffer.

`SJC$_NO_PROCESSOR` is a Boolean item code. It specifies that the symbiont image to be executed is `SYS$SYSTEM:PRTSMB.EXE`. It is the default.

SJC\$_PROTECTION

`SJC$_PROTECTION` is an input value item code. It specifies the protection of a queue. The buffer must specify a longword in the format shown in the figure below.

System Service Descriptions

SSNDJBC

Value change enable																Protection value															
WORLD				GROUP				OWNER				SYSTEM				WORLD				GROUP				OWNER				SYSTEM			
D	E	W	R	D	E	W	R	D	E	W	R	D	E	W	R	D	E	W	R	D	E	W	R	D	E	W	R	D	E	W	R
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

ZK-1724-84

Bits 0 through 15 specify the protection value: the four types of access (read, write, execute, delete) to be granted to the four classes of user (system, owner, group, world). Set bits deny access and clear bits allow access.

Bits 16 through 31 enable or disable bits 0 through 15. When a bit in the second word is set, the corresponding bit in the first word will affect the queue protection. When a bit in the second word is clear, the corresponding bit in the first word is ignored.

By default, the queue protection is (S:E,O:D,G:R,W:W).

SJC\$_QUEMAN_RESTART

SJC\$_NO_QUEMAN_RESTART

SJC\$_QUEMAN_RESTART is a Boolean item code. It specifies that the job controller should automatically restart the queue manager when the job controller recovers from an internal fatal error. An internal fatal error causes the job controller to close the job queue manager file and stop the queue manager.

If you specify SJC\$_QUEMAN_RESTART, batch and output queues are restored to the states that existed prior to the job controller failure. The job controller opens the job queue manager file that was open when the job controller aborted. The system uses the default values of 100 for SJC\$_EXTEND_QUANTITY and 50 for SJC\$_BUFFER_COUNT, rather than the values you may have specified for these item codes when you last started the queue manager with a SJC\$_START_QUEUE_MANAGER operation.

Note: To prevent a looping condition, the job controller does not restart the queue manager if it detects a job controller error within 2 minutes of starting the queue manager. This algorithm may change in a future release of VAX/VMS.

SJC\$_NO_QUEMAN_RESTART is a Boolean item code. It specifies that the job controller should not restart the queue manager when it recovers from an internal job controller fatal error. In this case, a user with OPER privilege must restart the queue manager and restore the queuing environment. It is the default.

SJC\$_QUEUE

SJC\$_QUEUE is an input value item code. It specifies the queue to which the operation is directed. The buffer must specify the name of the queue.

The string may contain uppercase or lowercase characters (lowercase are converted to uppercase), numeric characters, dollar signs, and underscores. If the string is a logical name, SYS\$SNDJBC translates it iteratively until the equivalence string is found or the number of translations allowed by the system has been performed. The maximum length of the final character string is 31 characters; spaces, tabs, and null characters are ignored.

SJC\$_QUEUE_FILE_SPECIFICATION

SJC\$_QUEUE_FILE_SPECIFICATION is an input value item code. It specifies the file specification of the system job queue file. The buffer must contain a valid RMS file specification. Omitted fields in the file specification are supplied from the default file specification SYS\$SYSTEM:JBCSYSQUE.DAT.

SJC\$_RECORD_BLOCKING

SJC\$_NO_RECORD_BLOCKING

SJC\$_RECORD_BLOCKING is a boolean item code. It is meaningful only for output execution queues. It specifies that the symbiont can concatenate the output records it sends to the output device. For the standard VMS print symbiont, record blocking can have a significant performance advantage over single-record mode. It is the default.

SJC\$_NO_RECORD_BLOCKING is a boolean item code. It specifies that the symbiont must send each record in a separate I/O request to the output device.

SJC\$_RELATIVE_PAGE

SJC\$_RELATIVE_PAGE is an input value item code. It is meaningful only for output execution queues. The buffer must specify a signed longword integer. This item code specifies that printing should be resumed after spacing forward (if the buffer value is positive) or backward (if the buffer value is negative) the specified number of pages.

SJC\$_REQUEUE

SJC\$_REQUEUE is a Boolean item code. It specifies that a job is to be requeued. By default, the job is deleted.

SJC\$_RESTART

SJC\$_NO_RESTART

SJC\$_RESTART is a Boolean item code. It specifies that a job can restart after a system failure or can be requeued during execution. It is the default for print jobs.

SJC\$_NO_RESTART is a Boolean item code. It specifies that a job cannot restart after a system failure or after a requeue operation. It is the default for batch jobs.

SJC\$_RETAIN_ALL_JOBS

SJC\$_RETAIN_ERROR_JOBS

SJC\$_NO_RETAIN_JOBS

SJC\$_RETAIN_ALL_JOBS is a Boolean item code. It specifies that jobs be retained in the queue with a completion status after they have been executed.

System Service Descriptions

SSNDJBC

SJC\$_RETAIN_ERROR_JOBS is a Boolean item code. It specifies that jobs be retained only if the job completed unsuccessfully (the job's completion status has the low bit clear).

SJC\$_NO_RETAIN_JOBS is a Boolean item code. It specifies that jobs are not to be retained in the queue after they have completed. It is the default.

SJC\$_SCSNODE_NAME

SJC\$_SCSNODE_NAME is an input value item code. It is meaningful only for execution queues in a VAXcluster environment. It specifies the VAX node on which a queue is to execute. The buffer must specify a 6-byte node name that matches the value of the SYSGEN parameter SCSNODE in effect on the target node. By default, the queue executes on the VAX node from which the queue is first started.

SJC\$_SEARCH_STRING

SJC\$_SEARCH_STRING is an input value item code. It is meaningful only for output execution queues. The buffer must specify a string of no more than 63 characters. This item code specifies that printing is to resume at the page containing the first occurrence of the specified string. The search for the string proceeds in the forward direction.

SJC\$_SWAP

SJC\$_NO_SWAP

SJC\$_SWAP is a Boolean item code. It is meaningful only for batch execution queues. It specifies that jobs initiated from a queue can be swapped. It is the default.

SJC\$_NO_SWAP is a Boolean item code. It specifies that jobs cannot be swapped.

SJC\$_TERMINAL

SJC\$_NO_TERMINAL

SJC\$_TERMINAL is a Boolean item code. It is meaningful only for output queues. It designates the queue type as terminal rather than printer.

SJC\$_NO_TERMINAL is a Boolean item code. It designates the queue type as printer rather than terminal.

SJC\$_TOP_OF_FILE

SJC\$_TOP_OF_FILE is a Boolean item code. It is meaningful only for output queues. It specifies that printing is to be resumed at the beginning of the file.

SJC\$_UIC

SJC\$_UIC is an input value item code. This value specifies the 4-byte UIC of the user on behalf of whom the request is made. By default, the UIC is taken from the requesting process.

This item code requires CMKRNL privilege.

SJC\$_USERNAME

SJC\$_USERNAME is an input value item code. This value specifies the 12-byte username of the user on behalf of whom the request is made. By default, the username is taken from the requesting process.

This item code requires CMKRNL privilege.

System Service Descriptions

SSNDJBC

SJC\$_WSDEFAULT **SJC\$_NO_WSDEFAULT**

SJC\$_WSDEFAULT is an input value item code. It is meaningful only for batch jobs and execution queues. It specifies the default working set size for batch jobs or jobs initiated from a batch queue, or the default working set size of a symbiont process connected to an output queue. A symbiont process can control several output queues; however, the default working set size of the symbiont process is established by the first queue to which it is connected. The buffer must contain a longword integer value in the range 1 through 65,535.

SJC\$_NO_WSDEFAULT is a Boolean item code. It specifies that the system determine the working set default. It is the default.

For batch jobs, The default working set size, working set quota, and working set extent (maximum size) are included in each user record in the system user authorization file (UAF). Values for these items can be specified for individual jobs and/or for all jobs in a given queue. Table SYS-7 shows the action taken when a value is specified for SJC\$_WSDEFAULT.

Table SYS-7 Working Set Decision Table

Value Specified for Job?	Value Specified for Queue?	Action Taken
No	No	Use UAF value
No	Yes	Use value for queue
Yes	Yes	Use lower of the two
Yes	No	Compare specified value with UAF value; use lower

SJC\$_WSEXTENT **SJC\$_NO_WSEXTENT**

SJC\$_WSEXTENT is an input value item code. It is meaningful only for batch jobs and execution queues. It specifies the working set extent for batch jobs or jobs initiated from a batch queue, or the working set extent of a symbiont process connected to an output queue. A symbiont process can control several output queues; however, the working set extent of the symbiont process is established by the first queue to which it is connected. The buffer must contain a longword integer value in the range 1 through 65,535.

SJC\$_NO_WSEXTENT is a Boolean item code. It specifies that the system determine the working set extent. It is the default.

Refer to the description of the SJC\$_WSDEFAULT item code and to Table SYS-7 for information on the action taken when a value is specified for SJC\$_WSEXTENT for a batch job.

SJC\$_WSQUOTA **SJC\$_NO_WSQUOTA**

SJC\$_WSQUOTA is an input value item code. It is meaningful only for batch jobs and execution queues. It specifies the working set quota for batch jobs or jobs initiated from a batch queue, or the working set quota of a symbiont process connected to an output queue. A symbiont process can control several output queues; however, the working set quota of the symbiont process is established by the first queue to which it is connected. The buffer must contain a longword integer value in the range 1 through 65,535.

System Service Descriptions

\$SNDJBC

SJC\$_NO_WSQUOTA is a Boolean item code. It specifies that the system determine the working set quota. It is the default.

Refer to the description of the SJC\$_WSDEFAULT item code and to Table SYS-7 for information on the action taken when a value is specified for SJC\$_WSQUOTA for a batch job.

iosb

VMS Usage: **io_status_block**
type: **quadword (unsigned)**
access: **write only**
mechanism: **by reference**

I/O status block into which \$SNDJBC writes the completion status after the requested operation has completed. The **iosb** is the address of the I/O status block.

At request initiation, \$SNDJBC sets the value of the quadword I/O status block to 0. When the requested operation has completed, \$SNDJBC writes a condition value in the first longword of the I/O status block. It writes the value 0 into the second longword; this longword is unused and reserved for future use.

The condition values returned by \$SNDJBC in the I/O status block are usually condition values from the JBC facility. These condition values are defined by the \$JBCMSGDEF macro. In some cases, the condition value returned by \$SNDJBC may be an error return from a system service or an RMS service that is used in executing the request. For the SJC\$_SYNCHRONIZE_JOB request, the condition value returned is the completion status of the requested job.

The condition values returned from the JBC facility are listed under the heading "Condition Values Returned in the I/O Status Block" after the Description section.

Though this argument is optional, DIGITAL strongly recommends that you specify it for the following reasons.

- If you are using an event flag to signal the completion of the service, you can test the I/O status block for a condition value to be sure that the event flag was not set by an event other than service completion.
- If you are using the \$SYNCH service to synchronize completion of the service, the I/O status block is a required argument for \$SYNCH.
- The condition value returned in R0 and the condition value returned in the I/O status block provide information about different aspects of the call to the \$SNDJBC service. The condition value returned in R0 gives you information about the success or failure of the service call itself; the condition value returned in the I/O status block gives you information about the success or failure of the service operation. Therefore, to accurately assess the success or failure of the call to \$SNDJBC, you must check the condition values returned in both R0 and the I/O status block.

astadr

VMS Usage: **ast_procedure**
type: **procedure entry mask**
access: **call without stack unwinding**
mechanism: **by reference**

AST service routine to be executed when \$SNDJBC completes. The **astadr** argument is the address of the entry mask of this routine.

System Service Descriptions

\$SNDJBC

If specified, the AST routine executes at the same access mode as the caller of \$SNDJBC.

astprm

VMS Usage: **user_arg**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

AST parameter to be passed to the AST service routine specified by the **astadr** argument. The **astprm** argument is this longword parameter.

DESCRIPTION

Types of Queue

The VAX/VMS batch/print facility supports several types of queue, which aid in the processing of batch and print jobs. The different types of queue can be divided into three major categories according to the way the system processes the jobs assigned to the queue. The three types of queue are: execution, generic, and logical. Execution queues schedule jobs for execution; generic and logical queues transfer jobs to execution queues. Within these major classifications, queue type is further defined by the kinds of job the queue can accept for processing. Some types of execution and generic queue accept batch jobs; other types accept print jobs. Logical queues are restricted to print jobs.

You create a queue by making a call to \$SNDJBC specifying the SJC\$_CREATE_QUEUE function code. Item codes that you optionally specify in the call determine the type of queue you create. The following list describes the various types of execution, generic, and logical queues and indicates which item codes you need to specify to create them.

- 1 **Execution queue.** An execution queue schedules jobs for processing. In a VAXcluster environment, jobs are processed on the node that manages the execution queue. The two types of execution queue, batch execution queue and output execution queue, are described below.
 - a **Batch execution queue.** A batch execution queue can schedule only batch jobs for execution. A batch job executes as a detached process that sequentially runs one or more command procedures; you define the list of command procedures as part of the initial job description. You create a batch execution queue by specifying the SJC\$_BATCH item code in the call to the \$SNDJBC service.
 - b **Output execution queue.** An output execution queue schedules print jobs for processing by an independent symbiont process that is associated with the queue. The job controller sends the symbiont a list of files to process; you define this list of files as part of the initial job description. As the symbiont processes each file, it produces output for the device it controls, such as a printer or terminal.

The standard print symbiont image provided by VAX/VMS is designed to print files on hardcopy devices. User-modified or user-written symbionts also can be designed for this or any other file processing activity managed by the VAX/VMS batch/print facility. The symbiont image that executes jobs from an output queue is specified by the SJC\$_PROCESSOR item code. If you omit this item code, the standard VAX/VMS print symbiont image, PRTSMB, is associated with the queue.

System Service Descriptions

\$SNDJBC

The three types of output execution queue are described below.

- **Printer execution queue.** This type of queue typically uses the standard print symbiont to direct output to a line printer. You can specify a user-provided symbiont in the `SJC$_PROCESSOR` item code. A printer execution queue is the default type of output execution queue.
- **Terminal execution queue.** This type of queue typically uses the standard print symbiont to direct output to a terminal printer. You can specify a user-provided symbiont in the `SJC$_PROCESSOR` item code. You create a terminal execution queue by specifying the `SJC$_TERMINAL` item code when you create the output execution queue.
- **Server execution queue.** This type of queue uses the user-modified or user-written symbiont you specify in the `SJC$_PROCESSOR` item code to process the files that belong to jobs in the queue. You cannot mark an output execution queue as a server execution queue when you create it. Only the user-provided symbiont can specify that the queue that it is associated with should be marked as a server execution queue.

As stated above, when you create an output execution queue you can initially mark it as either a printer or terminal execution queue. However, when the queue is started, the symbiont process associated with the queue can change the queue type from the type designated at its creation to a printer, terminal, or server execution queue as described below.

- When an output execution queue that is associated with the standard VAX/VMS print symbiont is started, the symbiont determines whether it is controlling a printer or terminal. It communicates this information to the job controller. If necessary, the job controller then changes the type designation of the output execution queue.
 - When an output execution queue that is associated with a user-modified or user-written symbiont is started, the symbiont has the option of identifying the queue to the job controller as a server queue. If the user-written or user-modified symbiont does not notify the job controller that it wants to change the queue type designation, the output execution queue retains the queue type designation it received when it was created. That is, it remains marked as either a printer or terminal execution queue, even though it is associated with a nonstandard symbiont.
- 2 Generic queue.** A generic queue holds a job until an appropriate execution queue becomes available to initiate the job; the job controller then requeues the job to the available execution queue. In a VAXcluster environment, a generic queue can direct jobs to execution queues that are located on other nodes in the VAXcluster.

You create a generic queue by specifying the `SJC$_GENERIC_QUEUE` item code in the call to the \$SNDJBC service. You designate each execution queue that the generic queue can direct jobs to by specifying the `SJC$_GENERIC_TARGET` item code. Because a generic queue can direct jobs to more than one execution queue, you can specify the `SJC$_GENERIC_TARGET` item code up to 124 times in a single call to \$SNDJBC to define a complete set of execution queues for any generic queue. If you do not specify the `SJC$_GENERIC_TARGET` item code, the generic

System Service Descriptions

\$SNDJBC

queue directs jobs to any execution queue that is the same type of queue as the generic queue; that is, a generic batch queue will direct a job to any available batch execution queue, and so on. There is one exception, a generic queue will not direct work to any execution queue that was created in a call to \$SNDJBC that specified the SJC\$_NO_GENERIC_SELECTION item code.

The two types of generic queue, generic batch queue and generic output queue, are described below.

- a **Generic batch queue.** A generic batch queue can direct jobs only to batch execution queues. You create a generic batch queue by specifying both the SJC\$_GENERIC_QUEUE and SJC\$_BATCH item codes in the call to the \$SNDJBC service.
 - b **Generic output queue.** A generic output queue can direct jobs to any of the three types of output execution queue: printer, terminal, or server. It is possible to create a generic output queue that directs jobs to any combination of the three types of output execution queue. Typically, however, when you create a generic output queue, you specify a list of type-specific target queues. This way, the generic output queue directs jobs to a single type of output execution queue. Thus, you can control whether the jobs submitted to the generic output execution queue are output on a line printer or a terminal printer, or are sent to a server symbiont for processing. You create a generic output queue by specifying the SJC\$_GENERIC_QUEUE item code in the call to the \$SNDJBC service.
- 3 **Logical queue.** A logical queue performs the same function as a generic output queue, except that a logical queue can direct jobs to only a single printer, terminal, or server execution queue. A logical queue is simply an output queue that has been assigned to transfer its jobs to one execution queue.

To change an output queue into a logical queue, you make a call to the \$SNDJBC service while the output queue is in a stopped state. The call must specify the SJC\$_ASSIGN_QUEUE function code and the SJC\$_DESTINATION_QUEUE item code. Use the SJC\$_DESTINATION_QUEUE item code to specify the execution queue to which the logical queue should direct jobs. When the logical queue is started, it automatically requeues its jobs to the specified execution queue as that execution queue becomes available. You can change a logical queue back to its original output queue definition by specifying the SJC\$_DEASSIGN_QUEUE function code in a subsequent call to the \$SNDJBC service.

Queue and Job Protection

There are three aspects to queue protection:

- The queue has an associated UIC. When a queue is created, it is assigned a UIC by using the SJC\$_OWNER_UIC item code. If this item code is not specified, the queue is given the default UIC [1,4].
- A queue may be assigned a protection mask by specifying the SJC\$_PROTECTION item code. This protection mask specifies read, write, execute, and delete access for the four categories of user: owner, group, world, and system.

System Service Descriptions

\$SNDJBC

- Certain queue operations require that the caller of \$SNDJBC have certain privileges. The function codes that require privilege are listed under the heading "Privileges and Restrictions."

There is a single way to protect a job. When a job is submitted to a queue, it is assigned a UIC that is the same as the UIC of the process that is submitting the job, unless the SJC\$_UIC item code is specified to supply a different UIC.

For each requested operation, the \$SNDJBC service checks the UIC and privileges of the requesting process against the UIC of the queue, protection specified for the queue, and the privilege(s), if any, required for the operation. This checking is performed in a way similar to the way that the file system checks access to a file by comparing the owner UIC and protection of the file with the UIC and privileges of the requester.

Operations that apply to jobs are checked against (1) the R (read) and D (delete) protection specified for the queue in which the job is entered and (2) the owner UIC of the job. In general, R access to a job allows a user to determine that the job exists; D access to a job allows the user to affect the job.

Operations that apply to queues are checked against (1) the W (Write) and E (Execute) protection specified for the queue and (2) the owner UIC of the queue. In general, W access to a queue allows a user to submit jobs to the queue; E access to a queue allows a user to act as an operator for the queue, including the ability to affect jobs in the queue, to affect accounting, and to alter queues. OPER privilege grants E access to all queues.

Privileges and Restrictions

To specify the following function codes, the caller must have both OPER and SYSNAM privilege:

SJC\$_START_QUEUE_MANAGER
SJC\$_STOP_QUEUE_MANAGER

To specify the following function codes, the caller must have OPER privilege:

SJC\$_CREATE_QUEUE
SJC\$_DEFINE_CHARACTERISTIC
SJC\$_DEFINE_FORM
SJC\$_DELETE_CHARACTERISTIC
SJC\$_DELETE_FORM
SJC\$_DELETE_QUEUE
SJC\$_START_ACCOUNTING
SJC\$_STOP_ACCOUNTING

To specify the following function code, the caller must have (1) OPER privilege, (2) E access to the queue containing the specified job, or (3) R access to the specified job:

SJC\$_SYNCHRONIZE_JOB

To specify the following function codes, the caller must have (1) OPER privilege, (2) E access to the specified queue, or (3) W access to the specified queue:

SJC\$_CREATE_JOB
SJC\$_ENTER_FILE

System Service Descriptions

SSNDJBC

To specify the following function codes, the caller must have OPER privilege or E access to the specified queue(s):

SJC\$_ALTER_QUEUE
SJC\$_ASSIGN_QUEUE
SJC\$_DEASSIGN_QUEUE
SJC\$_MERGE_QUEUE
SJC\$_PAUSE_QUEUE
SJC\$_RESET_QUEUE
SJC\$_START_QUEUE
SJC\$_STOP_QUEUE

To specify the following function codes, the caller must have (1) OPER privilege, (2) E access to the queue containing the specified job, or (3) D (delete) access to the specified job:

SJC\$_ABORT_JOB
SJC\$_ALTER_JOB
SJC\$_DELETE_JOB

To specify the following function codes, no privilege is required:

SJC\$_ADD_FILE
SJC\$_BATCH_CHECKPOINT
SJC\$_CLOSE_DELETE
SJC\$_CLOSE_JOB
SJC\$_WRITE_ACCOUNTING

To specify a base priority (using the SJC\$_BASE_PRIORITY item code) higher than the base priority of the requesting process requires OPER or ALTPRI privilege.

To specify a scheduling priority (using the SJC\$_PRIORITY item code) higher than the value of the SYSGEN parameter MAXQUEPRI requires OPER or ALTPRI privilege.

To specify the following item codes, the caller must have OPER privilege:

SJC\$_PROTECTION
SJC\$_OWNER_UIC

To specify the following item codes, the caller must have CMKRNL privilege:

SJC\$_ACCOUNT_NAME
SJC\$_UIC
SJC\$_USERNAME

CONDITION VALUES RETURNED

SS\$_NORMAL
SS\$_ACCVIO

Successful completion.

The item list or input buffer cannot be read by the caller; or the return length buffer, output buffer, or status block cannot be written by the caller.

SS\$_BADPARAM

The function code is invalid; the item list contains an invalid item code; a buffer descriptor has an invalid length; or the reserved parameter has a nonzero value.

SS\$_DEVOFFLINE

The job controller process is not running.

System Service Descriptions

\$SNDJBC

SS\$_EXASTLM

The **astadr** argument was specified, and the process has exceeded its ASTLM quota.

SS\$_ILLEFC

The **efn** argument specifies an illegal event flag number.

SS\$_INSFMEM

The executive mode stack contains insufficient space to complete the request.

SS\$_MBFULL

The job controller mailbox is full.

SS\$_MBTOOSML

The mailbox message is too large for the job controller mailbox.

SS\$_UNASEFC

The **efn** argument specifies an unassociated event flag cluster.

CONDITION VALUES RETURNED IN THE I/O STATUS BLOCK

JBC\$_NORMAL

Normal successful completion.

JBC\$_DELACCESS

The file protection of the specified file, which was entered with the delete option, does not allow delete access to the caller.

JBC\$_DUPFORM

The specified form number is invalid because it is already defined; each form must have a unique form number.

JBC\$_EMPTYJOB

The open job cannot be closed because it contains no files.

JBC\$_EXECUTING

The parameters of the specified job cannot be modified because the job is currently executing.

JBC\$_INCDSTQUE

The type of the specified destination queue is inconsistent with the requested operation.

JBC\$_INCFORMPAR

The specified length, width, and margin parameters are inconsistent; the value of the difference between the top and bottom margin parameters must be less than the form length, and the difference between the left and right margin parameters must be less than the line width.

JBC\$_INCOMPLETE

The requested queue management operation cannot be executed because a previously requested queue management operation has not yet completed.

JBC\$_INCQUETYP

The type of the specified queue is inconsistent with the requested operation.

JBC\$_INVCHANAM

A specified characteristic name is not syntactically valid.

JBC\$_INVDSTQUE

The destination queue name is not syntactically valid.

JBC\$_INVFORNAM

The form name is not syntactically valid.

JBC\$_INVFUNCOD

The specified function code is invalid.

JBC\$_INVITMCOD

The item list contains an invalid item code.

JBC\$_INVPARLEN

The length of a specified string is outside the valid range for that item code.

System Service Descriptions

\$SNDJBC

JBC\$_INVPARVAL	A parameter value specified for an item code is outside the valid range for that item code.
JBC\$_INVQUENAM	The queue name is not syntactically valid.
JBC\$_JOBQUEDIS	The request cannot be executed because the system job queue manager has not been started.
JBC\$_JOBQUEENA	The system job queue manager cannot be started because it is already running.
JBC\$_MISREQPAR	An item code that is required for the specified function code has not been specified.
JBC\$_NODSTQUE	The specified destination queue does not exist.
JBC\$_NOOPENJOB	The requesting process did not open a job with the SJC\$_CREATE_JOB function.
JBC\$_NOPRIV	The queue protection denies access to the queue for the specified operation.
JBC\$_NOQUESPACE	The system job queue file was full and could not be extended.
JBC\$_NORESTART	The specified job cannot be requeued because it was not defined to be restartable.
JBC\$_NOSUCHCHAR	The specified characteristic does not exist.
JBC\$_NOSUCHFORM	The specified form does not exist.
JBC\$_NOSUCHJOB	The specified job does not exist.
JBC\$_NOSUCHQUE	The specified queue does not exist.
JBC\$_NOTASSIGN	The specified queue cannot be deassigned because it is not assigned.
JBC\$_QUENOTSTOP	The specified queue cannot be deleted because it is not in a stopped state.
JBC\$_REFERENCED	The specified queue cannot be deleted because of existing references by other queues or jobs.
JBC\$_STARTED	The specified queue cannot be started because it is already running.

EXAMPLE

```
! Declare system service related symbols
INTEGER*4      SYS$SNDJBCW,
2              STATUS
INCLUDE        '($SJCDEF)'
! Define item list structure
STRUCTURE      /ITMLST/
UNION
  MAP
    INTEGER*2  BUFLN, ITMCD
    INTEGER*4  BUFADR, RETADR
  END MAP
  MAP
    INTEGER*4  END_LIST
  END MAP
END UNION
END STRUCTURE
! Define I/O status block structure
STRUCTURE      /IOSBLK/
INTEGER*4      STS, ZEROED
END STRUCTURE
```

System Service Descriptions

\$SNDJBC

```
! Declare $SNDJBCW item list and I/O status block
RECORD /ITMLST/ SUBMIT_LIST(6)
RECORD /IOSBLK/ IOSB

! Declare variables used in $SNDJBCW item list
CHARACTER*9    QUEUE                /'SYS$BATCH'/
CHARACTER*23   FILE_SPECIFICATION  /'$DISK1:[COMMON]TEST.COM'/
CHARACTER*12   USERNAME             /'PROJ3036  '/
INTEGER*4      ENTRY_NUMBER

! Initialize item list for the enter file operation
SUBMIT_LIST(1).BUFLN = 9
SUBMIT_LIST(1).ITMCD = SJC$_QUEUE
SUBMIT_LIST(1).BUFADR = %LOC(QUEUE)
SUBMIT_LIST(1).RETADR = 0
SUBMIT_LIST(2).BUFLN = 23
SUBMIT_LIST(2).ITMCD = SJC$_FILE_SPECIFICATION
SUBMIT_LIST(2).BUFADR = %LOC(FILE_SPECIFICATION)
SUBMIT_LIST(2).RETADR = 0
SUBMIT_LIST(3).BUFLN = 12
SUBMIT_LIST(3).ITMCD = SJC$_USERNAME
SUBMIT_LIST(3).BUFADR = %LOC(USERNAME)
SUBMIT_LIST(3).RETADR = 0
SUBMIT_LIST(4).BUFLN = 0
SUBMIT_LIST(4).ITMCD = SJC$_NO_LOG_SPECIFICATION
SUBMIT_LIST(4).BUFADR = 0
SUBMIT_LIST(4).RETADR = 0
SUBMIT_LIST(5).BUFLN = 4
SUBMIT_LIST(5).ITMCD = SJC$_ENTRY_NUMBER_OUTPUT
SUBMIT_LIST(5).BUFADR = %LOC(ENTRY_NUMBER)
SUBMIT_LIST(5).RETADR = 0
SUBMIT_LIST(6).END_LIST = 0

! Call $SNDJBCW service to submit the batch job
STATUS = SYS$SNDJBCW (,
2      %VAL(SJC$_ENTER_FILE)..
2      SUBMIT_LIST,
2      IOSB..)
IF (STATUS) STATUS = IOSB.STS
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
END
```

The preceding FORTRAN program demonstrates the use of the \$SNDJBCW service to submit a batch job that will execute on behalf of another user. No log file is produced for the batch job. This program saves the job's entry number. CMKRNL privilege is required to run this program.

\$SNDJBCW—Send to Job Controller and Wait for Completion

The Send to Job Controller and Wait for Completion (\$SNDJBW) and \$GETQUI services together provide the user interface to the Job Controller (JBC) facility. The \$SNDJBW service allows the user to create, stop, and manage queues and the jobs in those queues. Queues may be generic, batch, execution, or output queues. Jobs may be batch or print jobs.

The \$SNDJBCW service queues a request to the Job Controller. For most operations, \$SNDJBCW completes synchronously; that is, it returns to the caller after the operation has completed. However, if the requested operation is a pause queue, stop queue, or abort job operation, \$SNDJBCW returns to the caller after queuing the request. There is no way to synchronize completion of these operations. Also, \$SNDJBCW does not wait for a job to complete before it returns to the caller; to synchronize completion of a job, the caller must specify the SJC\$_SYNCHRONIZE_JOB function code.

The \$SNDJBCW service is identical to the Send to Job Controller (\$SNDJBC) service except that \$SNDJBC completes asynchronously; the \$SNDJBC service returns to the caller immediately after queuing the request, without waiting for the operation to complete.

Refer to the documentation of the \$SNDJBC service for additional information about the \$SNDJBCW service.

The \$SNDJBC and \$SNDJBCW services supersede the Send Message to Symbiont Manager (\$SND SMB) and Send Message to Accounting Manager (\$SND ACC) services. New programs should be written using \$SNDJBC or \$SNDJBCW, instead of \$SND SMB or \$SND ACC, and old programs using \$SND SMB or \$SND ACC should be converted to use \$SNDJBC or \$SNDJBCW as convenient.

FORMAT

SYSSNDJBCW [*efn*] ,*func* [,*nullarg*] [,*itmlst*] [,*iosb*]
[,*astadr*] [,*astprm*]

\$SENDOPR—Send Message to Operator

The \$SENDOPR service performs the following functions:

- Sends a user request to operator terminals
- Sends a user cancellation-request to operator terminals
- Sends an operator reply to a user terminal
- Enables an operator terminal
- Displays the status of an operator terminal
- Initializes the operator log file

FORMAT **SY\$SENDOPR** *msgbuf* [,*chan*]

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS *msgbuf*

VMS Usage: **char_string**
type: **character-coded text string**
access: **read only**
mechanism: **by descriptor—fixed length string descriptor**

User buffer specifying the operation to be performed and information needed to perform that operation. The *msgbuf* is the address of a character string descriptor pointing to the buffer.

The format and contents of the buffer vary with the requested operation; however, the first byte in any buffer is the request code, which specifies the operation to be performed. The \$OPCMG macro defines the symbolic names for these request codes. The following list shows each operation that \$SENDOPR performs and the request code that specifies that operation:

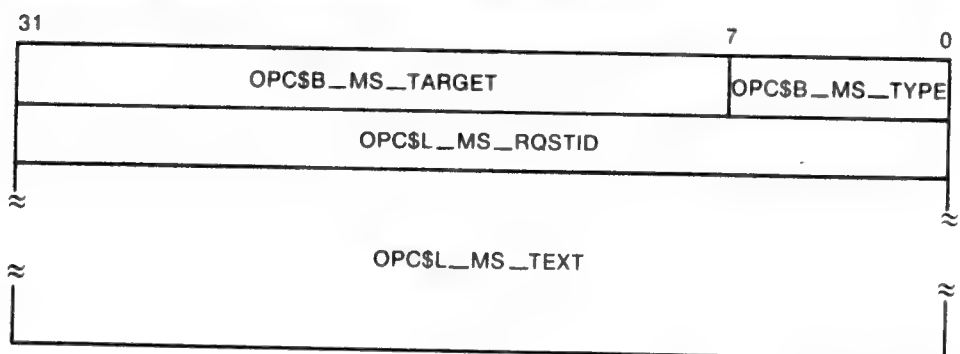
System Service Descriptions

\$SNDOPR

Request Code	Corresponding Operation
OPC\$_RQ_RQST	Sends a user request to operator terminals. This request code is used to make an operator request. If a reply to the request is specified (by using the chan argument), the operator request is kept active until the operator responds.
OPC\$_RQ_CANCEL	Sends a user cancellation-request to specified operator terminals. This request code is used to notify one or more operators that a previous request is to be cancelled. To specify OPC\$_RQ_CANCEL, the chan argument must also be specified.
OPC\$_RQ_REPLY	Sends an operator reply to a user who has made a request. This request code is used by operators to report the status of a user request. The format of the message buffer for this request is the format of the reply found in the user's mailbox after the call to \$SNDOPR completes. All functions of \$SNDOPR that deliver a reply to a mailbox do so in the format described for this request code.
OPC\$_RQ_TERME	Enables an operator terminal. This request is used to enable a specified terminal to receive operator messages.
OPC\$_RQ_STATUS	Reports the status of an operator terminal. This request is used by operators to display the operator classes for which the specified terminal is enabled and a list of outstanding requests.
OPC\$_RQ_LOGI	Initializes the operator log file.

What follows is a diagram of the message buffer for each of the above request codes. Each field within a diagram has a symbolic name, which serves to identify the field; in other words, these names specify offsets into the message buffer. The list following each diagram shows each field name and what its contents can or should be. The \$OPCDEF macro defines the field names, as well as any other symbolic name that may be specified as the contents of a field.

Message Buffer Format for OPC\$_RQ_RQST



ZK-1725-84

System Service Descriptions

SSNDOPR

OPC\$B_MS_TYPE

This 1-byte field contains the request code OPC\$_RO_RQST.

OPC\$B_MS_TARGET

This 3-byte field contains a 24-bit bit vector that specifies which operator terminal types are to receive the request. The \$OPCDEF macro defines symbolic names for the operator terminal types. The bit vector is constructed by specifying the desired symbolic names in a logical OR operation. The following list gives the symbolic name of each operator terminal type.

OPC\$_NM_CARDS	Card device operator
OPC\$_NM_CENTRL	Central operator
OPC\$_NM_CLUSTER	VAXcluster operator
OPC\$_NM_DEVICE	Device status information
OPC\$_NM_DISKS	Disk operator
OPC\$_NM_NETWORK	Network operator
OPC\$_NM_TAPES	Tape operator
OPC\$_NM_PRINT	Printer operator
OPC\$_NM_SECURITY	Security operator
OPC\$_NM_OPER1	OPC\$_NM_OPER1 through
to	OPC\$_NM_OPER12 specify
OPC\$_NM_OPER12	system-manager-defined operator functions.

OPC\$L_MS_RQSTID

This longword field contains a user-supplied longword message code.

OPC\$L_MS_TEXT

This variable-length field contains an ASCII string specifying text that is to be sent to the specified operator terminals. The length of the string must be in the range 0 to 255 bytes.

Message Buffer Format for OPC\$_RO_CANCEL

31	15	7	0
OPC\$B_MS_TARGET			OPC\$B_MS_TYPE
OPC\$L_MS_RQSTID			

ZK-1724-84

System Service Descriptions

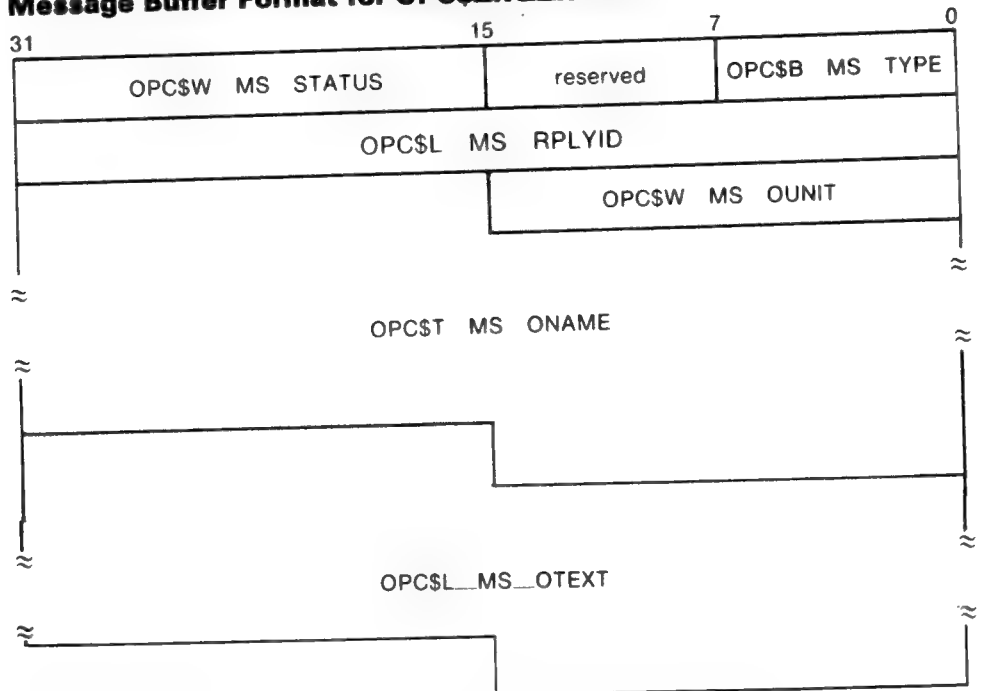
\$\$NDOPR

OPC\$B_MS_TYPE	This 1-byte field contains the request code OPC\$_RQ_CANCEL.																								
OPC\$B_MS_TARGET	<p>This 3-byte field contains a 24-bit bit vector that specifies which operator terminal types are to receive the cancellation request. The \$OPCDEF macro defines symbolic names for the operator terminal types. The bit vector is constructed by specifying the desired symbolic names in a logical OR operation. The following lists the symbolic name of each operator terminal type:</p> <table><tr><td>OPC\$_NM_CARDS</td><td>Card device operator</td></tr><tr><td>OPC\$_NM_CENTRL</td><td>Central operator</td></tr><tr><td>OPC\$_NM_SECURITY</td><td>Security operator</td></tr><tr><td>OPC\$_NM_CLUSTER</td><td>VAXcluster operator</td></tr><tr><td>OPC\$_NM_DEVICE</td><td>Device status information</td></tr><tr><td>OPC\$_NM_DISKS</td><td>Disk operator</td></tr><tr><td>OPC\$_NM_NETWORK</td><td>Network operator</td></tr><tr><td>OPC\$_NM_TAPES</td><td>Tape operator</td></tr><tr><td>OPC\$_NM_PRINT</td><td>Printer operator</td></tr><tr><td>OPC\$_NM_OPER1</td><td>OPC\$_NM_OPER1 through</td></tr><tr><td>to</td><td>OPC\$_NM_OPER12 specify</td></tr><tr><td>OPC\$_NM_OPER12</td><td>system-manager-defined operator functions.</td></tr></table>	OPC\$_NM_CARDS	Card device operator	OPC\$_NM_CENTRL	Central operator	OPC\$_NM_SECURITY	Security operator	OPC\$_NM_CLUSTER	VAXcluster operator	OPC\$_NM_DEVICE	Device status information	OPC\$_NM_DISKS	Disk operator	OPC\$_NM_NETWORK	Network operator	OPC\$_NM_TAPES	Tape operator	OPC\$_NM_PRINT	Printer operator	OPC\$_NM_OPER1	OPC\$_NM_OPER1 through	to	OPC\$_NM_OPER12 specify	OPC\$_NM_OPER12	system-manager-defined operator functions.
OPC\$_NM_CARDS	Card device operator																								
OPC\$_NM_CENTRL	Central operator																								
OPC\$_NM_SECURITY	Security operator																								
OPC\$_NM_CLUSTER	VAXcluster operator																								
OPC\$_NM_DEVICE	Device status information																								
OPC\$_NM_DISKS	Disk operator																								
OPC\$_NM_NETWORK	Network operator																								
OPC\$_NM_TAPES	Tape operator																								
OPC\$_NM_PRINT	Printer operator																								
OPC\$_NM_OPER1	OPC\$_NM_OPER1 through																								
to	OPC\$_NM_OPER12 specify																								
OPC\$_NM_OPER12	system-manager-defined operator functions.																								
OPC\$L_MS_RQSTID	This longword field contains a user-supplied longword message code.																								

System Service Descriptions

\$SNDOPR

Message Buffer Format for OPC\$_RQ_REPLY



ZK-1727-84

OPC\$B_MS_TYPE

This 1-byte field contains the request code OPC\$_RQ_REPLY.

reserved

This 1-byte field is reserved for future use.

OPC\$W_MS_STATUS

This 2-byte field contains the low-order word of the longword condition value that \$SNDOPR returns in the mailbox specified by the **chan** argument. A list of these longword condition values appears in the section entitled "Condition Values Returned in the Mailbox." To test the completion status, it is necessary to extract the low-order word from the longword condition value and compare it to the contents of the OPC\$W_MS_STATUS field.

OPC\$L_MS_RPLYID

This 4-byte field contains a user-supplied message code

System Service Descriptions

\$SNDOPR

OPC\$W_MS_OUNIT

This 2-byte field contains the unit number of the terminal to which the operator reply is to be sent. To obtain the unit number of the terminal, you can call \$GETDVI, specifying the DVI\$_FULLDEVNAM item code. The information returned will consist of the node name and device name as a padded string. Since the unit number is found on the tail end of the string, you must parse the string. One way to do this is, starting from the tail end, to search for the first alphabetic character; the digits to the right of this alphabetic character constitute the unit number.

After extracting the unit number, count the remaining characters in the string. Then, construct a counted ASCII string and use this for the following field, OPC\$T_MS_ONAME.

OPC\$T_MS_ONAME

This variable-length field contains a counted ASCII string specifying the device name of the terminal that is to receive the operator reply. The maximum total length of the string is 14 bytes. See the preceding field description (OPC\$T_MS_OUNIT) to learn how to obtain the device name.

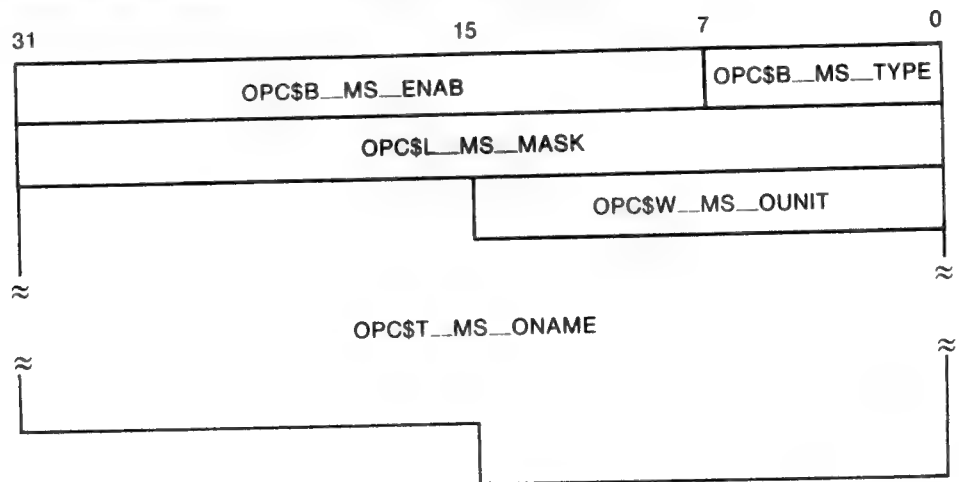
OPC\$L_MS_OTEXT

This variable-length field contains an ASCII string specifying operator-written text that is to be sent to the user terminal. The length of the string must be in the range 0 to 255 bytes. This field is optional.

System Service Descriptions

\$SNDOPR

Message Buffer Format for OPC\$_RQ_TERME



ZK-1728-84

OPC\$_B_MS_TYPE

This 1-byte field contains the request code OPC\$_RQ_TERME.

OPC\$_B_MS_ENAB

This 3-byte field contains a user-supplied value. The value 0 specifies that the specified terminal is to be disabled for the specified operator classes. Any nonzero value specifies that the specified terminal is to be enabled for the specified operator classes.

OPC\$_B_MS_MASK

This 4-byte field contains a 4-byte bit vector that specifies which operator terminal types are to be enabled or disabled for the specified terminal. The \$OPCDEF macro defines symbolic names for the operator terminal types. The bit vector is constructed by specifying the desired symbolic names in a logical OR operation. The following list gives the symbolic name of each operator terminal type:

OPC\$_M_NM_CARDS	Card device operator
OPC\$_M_NM_CENTRL	Central operator
OPC\$_M_NM_SECURITY	Security operator
OPC\$_M_NM_CLUSTER	VAXcluster operator
OPC\$_M_NM_DEVICE	Device status information
OPC\$_M_NM_DISKS	Disk operator
OPC\$_M_NM_NETWORK	Network operator
OPC\$_M_NM_TAPES	Tape operator
OPC\$_M_NM_PRINT	Printer operator
OPC\$_M_NM_OPER1	OPC\$_M_NM_OPER1 through
to	OPC\$_M_NM_OPER12 specify
OPC\$_M_NM_OPER12	system-manager-defined operator functions.

System Service Descriptions

\$SNDOPR

OPC\$W_MS_OUNIT

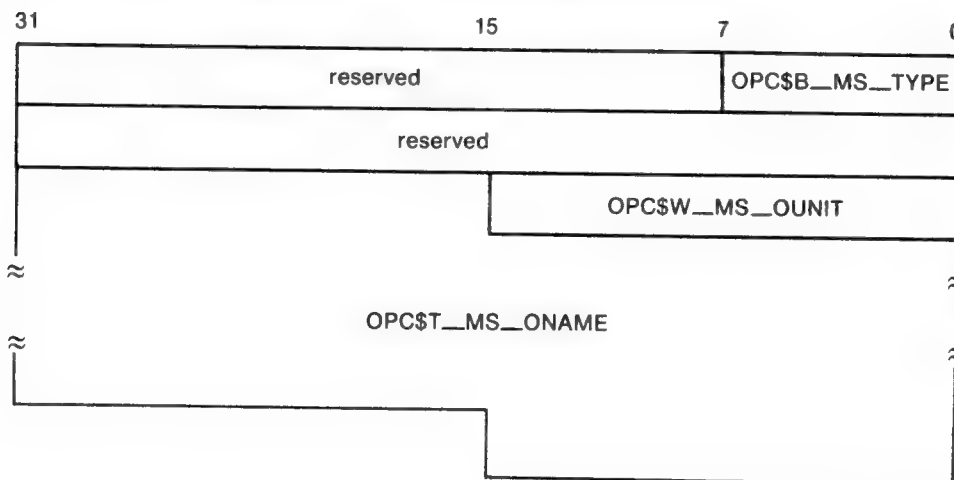
This 2-byte field contains the unit number of the operator terminal that is to be enabled or disabled for the specified operator terminal types. To obtain the unit number of the terminal, you can call \$GETDVI, specifying the DVI\$_FULLDEVNAM item code. The information returned will consist of the node name and device name as a padded string. Since the unit number is found on the tail end of the string, you must parse the string. One way to do this is, starting from the tail end, to search for the first alphabetic character; the digits to the right of this alphabetic character constitute the unit number.

After extracting the unit number, count the remaining characters in the string. Then, construct a counted ASCII string and use this for the following field, OPC\$_MS_ONAME.

OPC\$_MS_ONAME

This variable-length field contains a counted ASCII string specifying the device name of the operator terminal that is to be enabled or disabled for the specified operator terminal types. The maximum total length of the string is 16 bytes. See the preceding field description (OPC\$_MS_OUNIT) to learn how to obtain the device name.

Message Buffer Format for OPC\$_RQ_STATUS



ZK-1729-84

OPC\$B_MS_TYPE

This 1-byte field contains the request code OPC\$_RQ_STATUS.

Reserved

This 3-byte field is reserved for future use.

Reserved

This 4-byte field is reserved for future use.

System Service Descriptions

\$SNDOPR

OPC\$W_MS_OUNIT

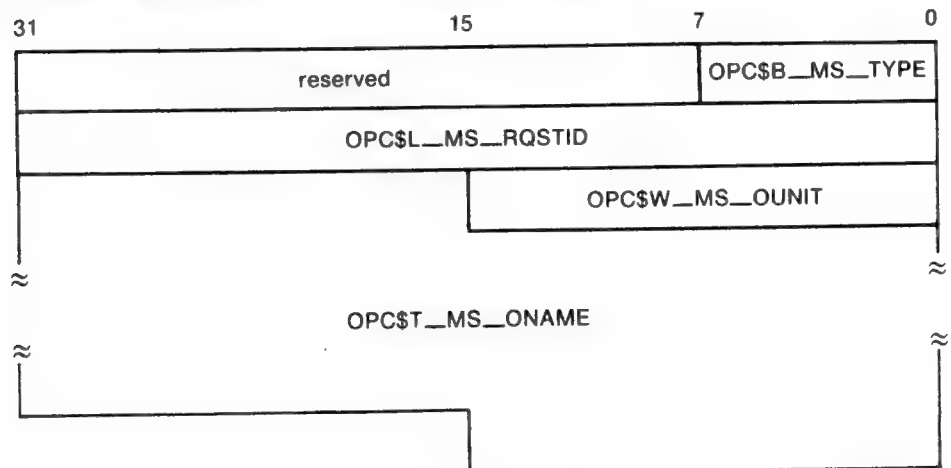
This 2-byte field contains the unit number of the operator terminal whose status is to be requested. To obtain the unit number of the terminal, you can call \$GETDVI, specifying the DVIS_FULLDEVNAM item code. The information returned will consist of the node name and device name as a padded string. Since the unit number is found on the tail end of the string, you must parse the string. One way to do this is, starting from the tail end, to search for the first alphabetic character; the digits to the right of this alphabetic character constitute the unit number.

After extracting the unit number, count the remaining characters in the string. Then, construct a counted ASCII string and use this for the following field, OPC\$T_MS_ONAME.

OPC\$T_MS_ONAME

This variable-length field contains a counted ASCII string specifying the device name of the operator terminal whose status is requested. The maximum total length of the string is 14 bytes. See the preceding field description (OPC\$T_MS_OUNIT) to learn how to obtain the device name.

Message Buffer Format for OPC\$_RQ_LOGI



OPC\$B_MS_TYPE

This 1-byte field contains the request code OPC\$_RQ_LOGI.

Reserved

This 3-byte field is reserved for future use.

OPC\$L_MS_RQSTID

This longword field contains a user-supplied value. The value 0 specifies that the current operator log file is to be closed and a new log file opened. The value 1 specifies that the current operator log file is to be closed but no new log file is to be opened.

System Service Descriptions

\$SENDOPR

OPC\$W_MS_OUNIT	<p>This 2-byte field contains the unit number of the operator terminal that is making the initialization request. To obtain the unit number of the terminal, you can call \$GETDVI, specifying the DVI\$_FULLDEVNAM item code. The information returned will consist of the node name and device name as a padded string. Since the unit number is found on the tail end of the string, you must parse the string. One way to do this is, starting from the tail end, to search for the first alphabetic character; the digits to the right of this alphabetic character constitute the unit number.</p> <p>After extracting the unit number, count the remaining characters in the string. Then, construct a counted ASCII string and use this for the following field, OPC\$T_MS_ONAME.</p>
OPC\$T_MS_ONAME	<p>This variable-length field contains a counted ASCII string specifying the device name of the operator terminal that is making the initialization request. The maximum total length of the string is 14 bytes. See the preceding field description (OPC\$T_MS_OUNIT) to learn how to obtain the device name.</p>

chan

VMS Usage: **channel**
type: **word (unsigned)**
access: **read only**
mechanism: **by value**

Channel assigned to the mailbox to which the reply is to be sent. The **chan** argument is a longword value containing the number of the channel. If **chan** is not specified or is specified as 0 (the default), no reply is sent.

The **chan** argument must be specified if a reply from the operator is desired.

DESCRIPTION

Depending on the operation, use of \$SENDOPR may require the calling process to have OPER privilege in order to perform the following functions:

- Enable a terminal as an operator's terminal
- Reply to or cancel a user's request
- Initialize the operator communication log file

In addition, the operator must have SECURITY privilege, as well as OPER privilege to affect security functions.

The Send Message To Operator system service requires system dynamic memory.

The general procedure for using this service is as follows:

- 1 Construct the message buffer and place its final length in the first word of the buffer descriptor.
- 2 Call the \$SENDOPR service.
- 3 Check the condition value returned in R0 to ensure the request was successfully made.
- 4 Issue a read request to the mailbox specified, if any.

System Service Descriptions

SSNDOPR

- 5 When the read completes, check the 2-byte condition value in the OPC\$W_MS_STATUS field to ensure that the operation was successfully performed.

The format of messages displayed on operator terminals follows:

```
XXXXXXXXXX OPCOM dd-mm-yyyy hh:mm:ss.cc
message specific information
```

The following example shows the message displayed on a terminal as a result of a request to enable that terminal as an operator terminal:

```
XXXXXXXXXX OPCOM 3-OCT-1983 13:44:40.37
Operator _NODE$LTAS: has been enabled, username HOEBLE
```

The following example shows the message displayed on an operator terminal as a result of a request to display the status of that operator terminal:

```
XXXXXXXXXX OPCOM 3-OCT-1983 12:11:10.48
Operator status for operator _NODE$OPAO:
CENTRAL, PRINTER, TAPES, DISKS, DEVICES, CARDS, CLUSTER, SECURITY,
OPER1, OPER2, OPER3, OPER4, OPER5, OPER6, OPER7, OPER8, OPER9,
OPER10, OPER11, OPER12
```

The following example shows the message displayed on an operator terminal as a result of a user request:

```
XXXXXXXXXX OPCOM 3-OCT-1983 12:57:32.25
Request 1285, from user ROSS on NODE_NAME
Please mount device _NODE$DMAO:
```

CONDITION VALUES RETURNED

SS\$_NORMAL	Service successfully completed.
SS\$_ACCVIO	The message buffer or buffer descriptor cannot be read by the caller.
SS\$_BADPARAM	The specified message has a length of 0 or has more than 986 bytes.
SS\$_DEVNOTMBX	The channel specified is not assigned to a mailbox.
SS\$_INSFMEM	Insufficient system dynamic memory is available to complete the service.
SS\$_IVCHAN	An invalid channel number was specified. An invalid channel number is one that is 0 or a number larger than the number of channels available.
SS\$_NOPRIV	The process does not have the privilege to reply to or cancel a user's request; the process does not have read/write access to the specified mailbox; or the channel was assigned from a more privileged access mode.

CONDITION VALUES RETURNED IN THE MAILBOX

OPC\$_BLANKTAPE	Successful completion; the operator responded with the DCL command REPLY/BLANK_TAPE=n.
OPC\$_INITAPE	Successful completion; the operator responded with the DCL command REPLY/INITIALIZE_TAPE=n.

System Service Descriptions

\$SENDOPR

OPC\$_NOOPERATOR	Successful completion; no operator terminal was enabled to receive the message.
OPC\$_RQSTCPLTE	Successful completion; the operator completed the request.
OPC\$_RQSTPEND	Successful completion; the operator will perform the request when possible.
OPC\$_RQSTABORT	The operator could not satisfy the request.
OPC\$_RQSTCAN	The caller canceled the request.

EXAMPLES

```

1  ;++
; Build and send an operator request.
;--

        $dscdef                ; Define descriptor offsets
        $opcdef                ; Define OPCOM message offsets
                                ; and codes
        $opcmsg                ; Define message type codes

; Local storage and data
;
bufsiz = <opc$L_ms_text+120>    ; Maximum request buffer size
rqstprmt:                ; Prompt for user request
        .ascid /Request> /
rqst:                ; User request text
                ; (dynamic string)
        .word 0
        .byte dsc$k_dtype_t
        .byte dsc$k_class_d
        .long 0
msgdesc:                ; Descriptor of request
                ; message buffer
        .long bufsiz
        .address msgbuf
msgbuf:                ; Request message buffer
        .blkb bufsiz
rqstid:                ; User request id number
        .long 0
        .page
        .sbt1 Main routine
;+
; Prompt user for request text.
;
; Build the request message.
;
; Send the request to the operator.
;-
        .entry oprexample,`m<r2,r3,r4,r5,r6,r7,r8,r9,r10,r11>
; Prompt user for request text.
;
        movaq rqstprmt,r2        ; get address of prompt string
        movaq rqst,r3           ; get address of result buffer desc.
prompt: pushaq (r2)              ; prompt string
        pushaq (r3)              ; result buffer
        calls #2,g`lib$get_input ; get the request text
        blbs r0,10$             ; branch if success
        ret                     ; return error status
10$:   tstw dsc$w_length(r3)      ; check for text
        beql prompt             ; branch if none - try again
; Build the request message.
;

```

System Service Descriptions

\$\$SENDOPR

```

movab    msgbuf,r4          ; get address
movb     #opc$rq_rqst,-      ; insert message type
;
insv     #opc$m_nm_disks,-   ; insert target mask (disks)
;         #0,-               ; starting at bit 0
;         #24,-              ; continue for 24 bits
;         opc$b_ms_target(r4); into the TARGET field
;
moval    rqstid,r5           ; get address of request id
incl     (r5)                ; set to next request number
movl     (r5),opc$L_ms_rqstid(r4); insert request number
pushr    #~m<r2,r3,r4,r5>    ; save registers
movc5    dsc$w_length(r3),-   ; copy request text
;         @dsc$a_pointer(r3),-; to message buffer
;         #0,-               ; fill with zeros
;         #120,-             ; truncate to 120 characters
;         opc$L_ms_text(r4)   ;
;
popr     #~m<r2,r3,r4,r5>    ; restore registers
movaq    msgdsc,r6           ; get address of
;         ; message descriptor
;
addw3    #opc$L_ms_text,-     ; calculate message length
;         dsc$w_length(r3),-  ;
;         dsc$w_length(r6)    ;
;
; Send the request to the operator.
;
$sendopr_s msgdsc            ; Send request
;                             ; (no reply expected)
ret                          ; return to caller
;
.end    oprexample

```

The VAX MACRO example above allows the user to build an operator request and send the request to the operator.

2

IMPLICIT NONE

```

! symbol definitions
INCLUDE '($DVIDEF)'
INCLUDE '($OPCDEF)'

! structures for SENDOPR
STRUCTURE /MESSAGE/
UNION
MAP
    BYTE TYPE,
    2    ENABLE(3)
    INTEGER*4 MASK
    INTEGER*2 OUNIT
    CHARACTER*14 ONAME
END MAP
MAP
    CHARACTER*24 STRING
END MAP
END UNION
END STRUCTURE
RECORD /MESSAGE/ MSGBUF
! length of MSGBUF.ONAME
INTEGER*4 ONAME_LEN

! status and routines
INTEGER*4 STATUS,
2    LIB$GETDVI,
2    SYS$SENDOPR

! type
MSGBUF.TYPE = OPC$rq_TERME
! enable
MSGBUF.ENABLE(1) = 1
! operator type
MSGBUF.MASK = OPC$m_nm_OPER1
! terminal unit number

```

System Service Descriptions

\$SENDOPR

```
STATUS = LIB$GETDVI (DVI$_UNIT,  
2  
2 'SYS$OUTPUT',  
2 MSGBUF.OUNIT,,)  
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))  
! terminal name  
STATUS = LIB$GETDVI (DVI$_FULLDEVNAM,  
2  
2 'SYS$OUTPUT',,  
2 MSGBUF.ONAME,  
2 ONAME_LEN)  
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))  
! remove unit number from ONAME and set up counted string  
ONAME_LEN = ONAME_LEN - 3  
MSGBUF.ONAME(2:ONAME_LEN+1) = MSGBUF.ONAME(1:ONAME_LEN)  
MSGBUF.ONAME(1:1) = CHAR(ONAME_LEN)  
! call $SENDOPR  
STATUS = SYS$SENDOPR (MSGBUF.STRING,)  
IF (.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))  
END
```

The VAX FORTRAN program above enables the current terminal to receive OPER1 operator messages.

\$SUSPND—Suspend Process

The Suspend Process service allows a process to suspend itself or another process. A suspended process cannot receive ASTs or otherwise be executed until another process resumes or deletes it.

FORMAT **SY\$SUSPND** [*pidadr*],[*prcnam*]

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS ***pidadr***

VMS Usage: **process_id**
type: **longword (unsigned)**
access: **modify**
mechanism: **by reference**

Process identification (PID) of the process to be suspended. The ***pidadr*** argument is the address of the longword PID.

The ***pidadr*** argument must be specified to suspend a process whose UIC group number is different than that of the calling process.

prcnam

VMS Usage: **process_name**
type: **character-coded text string**
access: **read only**
mechanism: **by descriptor—fixed length string descriptor**

Name of the process to be suspended. The ***prcnam*** argument is the address of a character string descriptor pointing to a 1- to 15-character process name string.

A process name is implicitly qualified by its UIC group number. Because of this, the ***prcnam*** argument can be used only to suspend processes in the same UIC group as the calling process.

To suspend processes in other groups, the ***pidadr*** argument must be specified.

If neither the ***pidadr*** nor ***prcnam*** arguments are specified, the caller process is suspended.

System Service Descriptions

\$SUSPND

DESCRIPTION Depending on the operation, use of \$SUSPN may require the calling process to have one of the privileges listed below:

- GROUP privilege is required to suspend another process in the same group, unless the process that is to be suspended has the same UIC as the calling process.
- WORLD privilege is required to suspend any other process in the system.

The \$SUSPND service requires system dynamic memory.

The \$SUSPND service completes successfully if the target process is already suspended.

Unless it has pages locked in the balance set, a suspended process can be removed from the balance set to allow other processes to execute.

The Resume Process (\$RESUME) service allows a suspended process to continue. If one or more resume requests are issued for a process that is not suspended, a subsequent suspend request completes immediately; that is, the process is not suspended. No count is maintained of outstanding resume requests.

CONDITION VALUES RETURNED

SS\$_NORMAL	Service successfully completed.
SS\$_ACCVIO	The process name string or string descriptor cannot be read by the caller, or the process identification cannot be written by the caller.
SS\$_INSFMEM	Insufficient system dynamic memory is available to complete the service.
SS\$_IVLOGNAM	The specified process name has a length of 0 or has more than 15 characters.
SS\$_NONEXPR	Warning; the specified process does not exist, or an invalid process identification was specified.
SS\$_NOPRIV	The target process was not created by the caller and the calling process does not have GROUP or WORLD privilege.

\$SYNCH—Synchronize

The Synchronize service checks the completion status of a system service that completes asynchronously.

The service whose completion status is to be checked must have been called with the **efn** and **iosb** arguments specified because the \$SYNCH service uses the event flag and I/O status block of the service to be checked.

Refer to Section 2.5 for a complete discussion of system service completion.

FORMAT	SY\$SYNCH [<i>efn</i>],[<i>iosb</i>]
---------------	---

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS *efn*

VMS Usage: **ef_number**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Number of the event flag specified in the call to the system service whose completion status is to be checked by \$SYNCH. The **efn** argument is a longword containing this number.

iosb

VMS Usage: **io_status_block**
type: **quadword (unsigned)**
access: **write only**
mechanism: **by reference**

I/O status block specified in the call to the system service whose completion status is to be checked by \$SYNCH. The **iosb** argument is the address of this quadword I/O status block.

DESCRIPTION

Before the implementation of the \$SYNCH service, users would test for the completion of an asynchronous system service (for example, \$GETJPI) by calling the \$WAITFR service. The \$WAITFR service would wait for the event flag specified in \$GETJPI to be set and would then return to the caller. When \$WAITFR returned, the user assumed that \$GETJPI had completed.

System Service Descriptions

\$SYNCH

This method never worked satisfactorily because the event flag could have been set by an event not associated with the completion of the \$GETJPI service. The true test for the completion of \$GETJPI is a nonzero I/O status block, since when \$GETJPI completes, it sets the event flag and writes a condition value in the I/O status block.

The \$SYNCH service performs this true test for the completion of an asynchronous service such as \$GETJPI. \$SYNCH operates in the following way:

- 1 When called, \$SYNCH waits (by calling the \$WAITFR service) for the event flag to be set.
- 2 When the event flag is set, \$SYNCH checks to see whether the I/O status block is nonzero. If it is nonzero, then the asynchronous service has truly completed, and \$SYNCH returns to the caller.
- 3 If the I/O status block is zero, then the asynchronous service has not yet completed and the event flag was set by the completion of an event not associated with the completion of \$GETJPI. In this case, \$SYNCH clears the event flag (by calling the \$CLREF service) and waits again (by calling \$WAITFR) for the event flag to be set, repeating this cycle until the I/O status block is nonzero.

The \$SYNCH service always sets the specified event flag when it returns to the caller. This ensures that different program segments can use the same event flag without clashing. For example, assume that calls to \$GETJPI and \$GETSYI both specify the same event flag and that \$SYNCH is called to check for the completion of \$GETJPI. If the event flag is set by \$GETSYI, \$SYNCH clears the flag and waits for it to be set by \$GETJPI. When \$GETJPI sets the flag, \$SYNCH returns to the caller and sets the event flag. In this way, the setting of the flag by \$GETSYI will not have been lost, and another call to \$SYNCH will show the completion of \$GETSYI.

The \$SYNCH service is useful when a program calls an asynchronous service but must perform some other work before testing for the completion of the asynchronous service. In this case, the program should call \$SYNCH at that point when it must know that the service has completed and when it is willing to wait for the service to actually complete.

When a program calls an asynchronous service (for example, \$QIO) and actually waits in-line (by calling \$WAITFR) for its completion without performing any other work, that program could be improved by calling the synchronous form of that service (for example, \$QIOW). The synchronous services such as \$QIOW execute code that checks for the true completion status in the same way that \$SYNCH does.

CONDITION VALUES RETURNED

SS\$_NORMAL

Service successfully completed. The asynchronous service has completed, and the I/O status block contains the condition value describing the completion status of the asynchronous service.

SYSS\$RMSRUNDWN—RMS Rundown

The RMS Rundown service closes all files opened by RMS for the image or process and halts I/O activity. This routine performs a \$CLOSE service for each file that is opened for processing.

FORMAT **SYSS\$RMSRUNDWN** *buf-addr, type-value*

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED".

ARGUMENTS ***buf-addr***

VMS Usage: **char_string**
type: **character-coded text string**
access: **write only**
mechanism: **by descriptor**

A descriptor pointing to a 22-byte buffer that will receive the device identification (16 bytes) and the file identification (6 bytes) of an improperly closed output file. The **buf-addr** argument is the address of the descriptor that points to the buffer.

type-value

VMS Usage: **byte_unsigned**
type: **byte (unsigned)**
access: **read only**
mechanism: **by value**

A single byte code that specifies the type of I/O rundown to be performed. The **type-value** argument is the actual value used.

This type code has the following values and meanings:

Arguments

0

Rundown of image and indirect I/O for process permanent files.

1

Rundown of image and process permanent files: the caller's mode must be other than user.

2

Abort RMS I/O: the caller's mode must be either executive or kernel (the system calls the I/O rundown control routine with this argument for process deletion).

System Service Descriptions

SYSSRMSRUNDWN

DESCRIPTION

In addition to closing all files and terminating I/O activity, the I/O rundown control routine will release all locks held on records in shared files, clear buffers, and return other resources allocated for file processing. You should continue to call the rundown control routine until you receive the success completion status code of RMS\$_NORMAL.

Note that prior to the execution of the \$CLOSE service, the rundown control routine will cancel all outstanding file operations specified in a FAB control block or any QIO requests related to file operations (an Open, Create, or Extend service, for example). It will also cancel any read/write requests to nondisk devices such as terminals or mailboxes prior to the execution of the Close service. All read/write requests of disk I/O buffers are, however, allowed to complete, which guarantees that none of the data written to disk files will be lost.

There is no predefined macro of the form \$RMSRUNDWN_G or \$RMSRUNDWN_S to call this service.

CONDITION VALUES RETURNED

RMS\$_NORMAL

Successful completion.

RMS\$_CCF

Cannot close file.

RMS\$_IAL

Invalid argument list. An output file could not be closed successfully, and the user buffer could not be written.

SYS\$SETDDIR—Set Default Directory

The Set Default Directory service allows you to read and/or change the default directory string for the process. You should restore the old default directory string to its original status unless you want the changed default directory string to last beyond the exit of your image.

FORMAT

SYSS\$SETDDIR *[new-dir-addr] [,length-addr]
[,cur-dir-addr]*

RETURNS

VMS Usage: cond_value
type: longword (unsigned)
access: write only
mechanism: by value

Longword condition value. All system services return (by value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS

new-dir-addr

VMS Usage: char_string
type: character-coded text string
access: read only
mechanism: by descriptor-fixed length s

A descriptor of the new default directory. The **new-dir-addr** argument is the address of the descriptor that points to the buffer containing the new directory specification that RMS will use to set the new process-default directory. If the default directory is not to be changed, the value of the **new-dir-addr** argument should be 0.

length-addr

VMS Usage: **word_unsigned**
type: **word (unsigned)**
access: **write only**
mechanism: **by reference**

length-addr A word that is to receive the length of the current default directory. The **length-addr** argument is the address of the word that will receive the length. Specify a value of 0 if you do not want this value returned.

cur-dir-addr

VMS Usage: **char_string**
type: **character-coded text string**
access: **write only**
mechanism: **by descriptor-fixed length string descriptor**

A descriptor of a buffer that is to receive the current default directory string. The **cur-dir-addr** argument is the address of the descriptor that points to the buffer area that is to receive the current directory string. Specify a value of 0 if you do not want this to be used.

System Service Descriptions

SYSS\$SETDDIR

DESCRIPTION The new directory name string is checked for correct syntax.
There is no predefined macro of the form \$SETDDIR_G or \$SETDDIR_S to call this service.

CONDITION	RMS\$_NORMAL	Successful completion
VALUES	RMS\$_DIR	Error in directory name
RETURNED	RMS\$_IAL	Invalid argument list

FORMAT **SYS\$SETDFPROT** [*new-def-prot-addr,*
 cur-def-prot-addr]

Longword condition value. All system services return (by value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

mechanism: by reference
A word that is to receive the current default file protection specification. The **cur-def-prot-addr** argument is the address of the word that receives the current process-default protection. Specify 0 if this is not wanted.

There is no predefined macro of the form `$SETDEFPROT_G` or `$SETDEFPROT_S` to call this service.

Successful completion
Invalid argument list

\$TRNLNM—Translate Logical Name

The Translate Logical Name service returns information about a logical name.

FORMAT

SYS\$TRNLNM *[attr],tabnam,lognam,[acmode]
[itmlst]*

RETURNS

VMS Usage: cond_value
type: longword (unsigned)
access: write only
mechanism: by value

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITIONS VALUES RETURNED."

ARGUMENTS *attr*

VMS Usage: mask_longword
type: longword (unsigned)
access: read only
mechanism: by reference

mechanism: by reference
Attributes controlling the search for the logical name. The **attr** argument is the address of a longword bit mask specifying these attributes.

Each bit in the longword corresponds to an attribute and has a symbolic name. These symbolic names are defined by the `$LNМDEF` macro. To specify an attribute, specify its symbolic name or set its corresponding bit. All undefined bits in the longword must be 0.

If this argument is not specified or is specified as 0 (no bits set), none of the attributes are used. The following lists the attributes:

Attribute	Description
LNMSM_CASE_BLIND	If set, \$TRNLNM does not distinguish between uppercase and lowercase letters in the logical name to be translated.

tabnam

VMS Usage: logical_name
type: character-coded text string
access: read only
mechanism: by descriptor—fixed length string descriptor

Name of the table or name of a list of table names in which to search for the logical name. The **tabnam** argument is the address of a descriptor pointing to this name. This argument is required.

System Service Descriptions

\$TRNLNM

If the table name is not the name of a logical name table, it is assumed to be a logical name and is translated iteratively until either the name of a logical name table is found or the number of translations allowed by the system have been performed. If the table name translates to a list of logical name tables, the tables are searched in the specified order.

lognam

VMS Usage: **logical_name**
type: **character-coded text string**
access: **read only**
mechanism: **by descriptor—fixed length string descriptor**

Logical name about which information is to be returned. The **lognam** argument is the address of a descriptor pointing to the logical name string. This argument is required.

acmode

VMS Usage: **access_mode**
type: **byte (unsigned)**
access: **read only**
mechanism: **by reference**

Access mode to be used in the translation. The **acmode** argument is the address of a byte specifying the access mode. The \$PSLDEF macro defines symbolic names for the four access modes.

When the **acmode** argument is specified, \$TRNLNM ignores all names (both logical names and table names) at access modes less privileged than the specified access mode. The specified access mode is not checked against that of the caller.

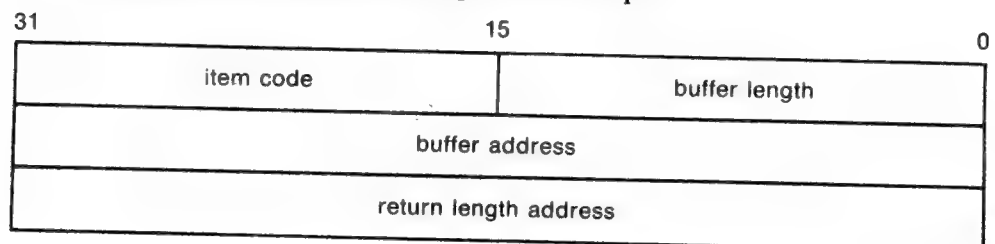
If **acmode** is not specified, \$TRNLNM performs the translation without regard to access mode; however, the translation process proceeds from the outermost to the innermost access modes. Thus, if two logical names with the same name, but at different access modes, exist in the same table, \$TRNLNM will translate the name with the outermost access mode.

itmlst

VMS Usage: **item_list_3**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Item list describing the information that \$TRNLNM is to return. The **itmlst** argument is the address of a list of item descriptors, each of which specifies or controls an item of information to be returned. The list of item descriptors is terminated by a longword of 0.

The following diagram depicts a single item descriptor.



ZK-1705-84

System Service Descriptions

\$TRNLNM

\$TRNLNM Item Descriptor Fields

buffer length

A word specifying the number of bytes in the buffer pointed to by the **buffer address** field.

item code

A word containing a symbolic code describing the nature of the information in the buffer or to be returned to the buffer pointed to by the **buffer address** field. Each item code is described below.

buffer address

A longword containing the address of the buffer that specifies or receives the information.

return length address

A longword containing the address of a word that specifies the actual length in bytes of the information returned by \$TRNLNM in the buffer pointed to by the **buffer address** field.

\$TRNLNM Item Codes

LNMS_ACMODE

When LNMS_ACMODE is specified, \$TRNLNM returns the access mode that was associated with the logical name at the time of its creation. The **buffer address** field in the item descriptor is the address of a byte in which \$TRNLNM writes the access mode.

LNMS_ATTRIBUTES

When LNMS_ATTRIBUTES is specified, \$TRNLNM returns the attributes of the logical name and the equivalence name associated with the current LNMS_INDEX value.

The **buffer address** field of the item descriptor points to a longword bit mask wherein each bit corresponds to an attribute. The \$TRNLNM service sets the corresponding bit for each attribute possessed by either the logical name or the equivalence name.

The symbolic names for these attributes are defined by the \$LNMDEF macro. The following list describes each attribute.

Attribute	Description
LNMSM_CONCEALED	If \$TRNLNM sets this bit, the equivalence name at the current index value for the logical name is a concealed logical name, as interpreted by RMS.
LNMSM_CONFINE	If \$TRNLNM sets this bit, the logical name is not copied from a process to any of its spawned subprocesses. The DCL SPAWN command creates subprocesses.
LNMSM_CRELOG	If \$TRNLNM sets this bit, the logical name was created using the \$CRELOG system service.
LNMSM_EXISTS	If \$TRNLNM sets this bit, an equivalence name with the specified index does exist.
LNMSM_NO_ALIAS	If \$TRNLNM sets this bit, the name of the logical name may not be given to another logical name defined in the same table at an outer access mode.

System Service Descriptions

\$TRNLNM

Attribute	Description
LNMSM_TABLE	If \$TRNLNM sets this bit, the logical name is the name of a logical name table.
LNMSM_TERMINAL	If \$TRNLNM sets this bit, the equivalence name for the logical name cannot be subjected to further (recursive) logical name translation.

LNMS_CHAIN

When LNMS_CHAIN is specified, \$TRNLNM processes another item list immediately following the current item list. LNMS_CHAIN must be the last item code in the current item list. The **buffer address** field of the item descriptor points to the next item list.

LNMS_INDEX

When LNMS_INDEX is specified, \$TRNLNM searches for an equivalence name having the specified index value. The **buffer address** field of the item descriptor points to a longword containing a user-specified integer in the range 0 to 127.

If this item code is not specified, the implied value of LNMS_INDEX is 0 and \$TRNLNM returns information about the equivalence name at index 0.

Since a logical name may have more than one equivalence name and each equivalence name is identified by an index value, you should specify the LNMS_INDEX item code first in the item list, before specifying LNMS_STRING, LNMS_LENGTH, or LNMS_ATTRIBUTES, since these item codes return information about the equivalence name identified by the current index value, LNMS_INDEX.

LNMS_LENGTH

When LNMS_LENGTH is specified, \$TRNLNM returns the actual length of the equivalence name string corresponding to the current LNMS_INDEX value. The **buffer address** field in the item descriptor is the address of the longword in which \$TRNLNM writes this length.

If an equivalence name does not exist at the current LNMS_INDEX value, \$TRNLNM returns a zero to the longword pointed to by the return length field of the item descriptor.

LNMS_MAX_INDEX

Each equivalence name for the logical name has an index associated with it. When LNMS_MAX_INDEX is specified, \$TRNLNM returns a value equal to the largest equivalence name index. The **buffer address** field in the item descriptor is the address of a longword in which \$TRNLNM writes this value. If no equivalence names (and, therefore, no index values) exist, \$TRNLNM returns a value of -1.

LNMS_STRING

When LNMS_STRING is specified, \$TRNLNM returns the equivalence name string corresponding to the current LNMS_INDEX value. The **buffer address** field of the item descriptor points to a buffer containing this string. The **return length address** field of the item descriptor contains an address of a word that contains the length in bytes of this string. The maximum length of the equivalence name string is 255 characters.

If an equivalence name does not exist at the current LNMS_INDEX value, \$TRNLNM returns the value zero in the **return length address** field of the item descriptor.

System Service Descriptions

\$TRNLNM

LNMS_TABLE

When **LNMS_TABLE** is specified, **\$TRNLNM** returns the name of the table containing the logical name being translated. The **buffer address** field of the item descriptor points to the buffer in which **\$TRNLNM** returns this name. The **return length address** field of the item descriptor specifies the address of a word in which **\$TRNLNM** writes the size of the table name. The maximum length of the table name is 31 characters.

DESCRIPTION Read access to a shareable logical name table is required to translate a logical name located in that shareable logical name table.

CONDITION VALUES RETURNED

SS\$_ACCVIO

The service cannot access the location(s) specified by one or more arguments.

SS\$_BADPARAM

One or more arguments has an invalid value, or a logical name table name or logical name was not specified.

SS\$_BUFFEROVF

Successful completion; the **buffer length** field in an item descriptor specified an insufficient value, so the buffer was not large enough to hold the requested data.

SS\$_IVLOGNAM

The **tabnam** argument or **lognam** argument specifies a string whose length is not in the required range of 1 through 255 characters.

SS\$_IVLOGTAB

The **tabnam** argument does not specify a logical name table.

SS\$_NOLOGNAM

The logical name was not found in the specified logical name table(s).

SS\$_NOPRIV

The caller lacks the necessary privilege to access the specified name.

SS\$_NORMAL

Successful completion; an equivalence name for the logical name has been found.

SS\$_TOOMANYLNAM

Logical name translation of the table name exceeded the allowable depth (10 translations).

\$ULKPAG—Unlock Pages from Memory

The Unlock Pages from Memory service unlocks pages that were previously locked in memory by the Lock Pages in Memory (\$LCKPAG) service.

Locked pages are automatically unlocked and deleted at image exit.

FORMAT **SY\$ULKPAG** *inadr* , [*retadr*] , [*acmode*]

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS *inadr*

VMS Usage: **address_range**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Starting and ending virtual addresses of the pages to be unlocked. The *inadr* is the address of a two-longword array containing, in order, the starting and the ending process virtual addresses. Only the virtual page number portion of each virtual address is used; the low-order 9 bits are ignored. If the starting and ending virtual addresses are the same, a single page is unlocked.

If more than one page is being unlocked and it is necessary to determine specifically which pages had been previously unlocked, the pages should be unlocked one at a time, that is, one page per call to \$ULWSET. The condition value returned by \$ULWSET will indicate whether the page was previously unlocked.

retadr

VMS Usage: **address_range**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference—array reference or descriptor**

Starting and ending process virtual addresses of the pages actually unlocked by \$ULKPAG. The *retadr* is the address of a two-longword array containing, in order, the starting and ending process virtual addresses.

If an error occurs while multiple pages are being unlocked, *retadr* specifies those pages that were successfully unlocked before the error occurred. If no pages were successfully unlocked, both longwords in the *retadr* array will contain the value -1.

System Service Descriptions

SULKPAG

acmode

VMS Usage: **access_mode**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Access mode on behalf of which the request is being made. The **acmode** argument is a longword containing the access mode. The \$PSLDEF macro defines the symbols for the four access modes.

The most privileged access mode used is the access mode of the caller. To unlock any specified page, the resultant access mode must be equal to or more privileged than the access mode of the owner of that page.

DESCRIPTION To call the \$ULKPAG service, a process must have PSWAPM privilege.

CONDITION VALUES RETURNED

SS\$_WASCLR

Service successfully completed. At least one of the specified pages was previously unlocked.

SS\$_WASSET

Service successfully completed. All of the specified pages were previously locked.

SS\$_ACCVIO

The input array cannot be read by the caller; the output array cannot be written by the caller; or a page in the specified range is inaccessible or does not exist.

System Service Descriptions

\$ULWSET

\$ULWSET—Unlock Pages from Working Set

The Unlock Pages from Working Set service unlocks pages that were previously locked in the working set by the Lock Pages in Working Set (\$LKWSET) service. Unlocked pages become candidates for replacement within the process's working set.

FORMAT

SYSS\$ULWSET *inadr* ,[*retadr*] ,[*acmode*]

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS

inadr

VMS Usage: **address_range**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference—array reference or descriptor**

Starting and ending virtual addresses of the pages to be unlocked. The *inadr* is the address of a two-longword array containing, in order, the starting and the ending process virtual addresses. Only the virtual page number portion of each virtual address is used; the low-order 9 bits are ignored. If the starting and ending virtual address are the same, a single page is unlocked.

If more than one page is being unlocked and it is necessary to determine specifically which pages had been previously unlocked, the pages should be unlocked one at a time, that is, one page per call to \$ULWSET. The condition value returned by \$ULWSET will indicate whether the page was previously unlocked.

retadr

VMS Usage: **address_range**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference—array reference or descriptor**

Starting and ending process virtual addresses of the pages that were actually unlocked by \$CRMPSC. The *retadr* is the address of a two-longword array containing, in order, the starting and ending process virtual addresses.

If an error occurs while multiple pages are being unlocked, *retadr* specifies those pages that were successfully unlocked before the error occurred. If no pages were successfully unlocked, both longwords in the *retadr* array will contain the value -1.

System Service Descriptions

\$ULWSET

acmode

VMS Usage: **access_mode**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Access mode on behalf of which the request is being made. The **acmode** argument is a longword containing the access mode. The \$PSLDEF macro defines the symbols for the four access modes.

The most privileged access mode used is the access mode of the caller. To unlock any specified page, the resultant access mode must be equal to or more privileged than the access mode of the owner of that page.

CONDITION VALUES RETURNED

SS\$_WASCLR

Service successfully completed. At least one of the specified pages was previously unlocked.

SS\$_WASSET

Service successfully completed. All of the specified pages were previously locked in the working set.

SS\$_ACCVIO

The **inadr** argument cannot be read by the caller; the **retadr** argument cannot be written by the caller; or a page in the specified range is inaccessible or does not exist.

SS\$_NOPRIV

A page in the specified range is in the system address space.

System Service Descriptions

\$UNWIND

\$UNWIND—Unwind Call Stack

The Unwind Call Stack service unwinds the procedure call stack; that is, it removes a specified number of call frames from the stack. Optionally, it may return control to a new PC after unwinding the stack. \$UNWIND is intended to be called from within a condition-handling routine.

FORMAT

SY\$UNWIND [*depadr*],[*newpc*]

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS

depadr

VMS Usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Depth to which the procedure call stack is to be unwound. The *depadr* argument is the address of a longword value. The value 0 specifies the call frame of the procedure that was executing when the condition occurred (that is, no call frames are unwound), 1 specifies the caller of that frame, 2 specifies the caller of the caller of that frame, and so on.

If *depadr* specifies the value 0, no unwind occurs and \$UNWIND returns a successful condition value in R0.

If *depadr* is not specified, \$UNWIND unwinds the stack to the call frame of the procedure that called the procedure that established the condition handler that is calling the \$UNWIND service. This is the default and the normal method of unwinding the procedure call stack.

newpc

VMS Usage: **address**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

New value for the program counter (PC); this value replaces the current value of the PC in the call frame of the procedure that receives control when the unwinding operation is complete. The *newpc* argument is a longword value containing the address at which execution is to resume.

Execution resumes at this address when the unwinding operation has completed.

System Service Descriptions

\$UNWIND

If **newpc** is not specified, execution resumes at the location specified by the PC in the call frame of the procedure that receives control when the unwinding operation is complete.

DESCRIPTION The actual unwind is not performed immediately. Rather, the return addresses in the call stack are modified so that when the condition handler returns, the unwind procedure is called from each frame that is being unwound.

During the actual unwinding of the call stack, \$UNWIND examines each frame in the call stack to see if a condition handler has been declared. If a handler has been declared, \$UNWIND calls the handler with the condition value **SS\$_UNWIND** (indicating that the call stack is being unwound) in the condition name argument of the signal array. When a condition handler is called with this condition value, that handler can perform any procedure-specific clean-up operations that may be required. After the condition handler returns, the call frame is removed from the stack.

CONDITION VALUES RETURNED

SS\$_NORMAL	Service successfully completed.
SS\$_ACCVIO	The call stack is not accessible to the caller. This condition is detected when the call stack is scanned to modify the return address.
SS\$_INSFRAME	There are insufficient call frames to unwind to the specified depth.
SS\$_NOSIGNAL	Warning. No signal is currently active for an exception condition.
SS\$_UNWINDING	Warning. An unwind operation is already in progress.

System Service Descriptions

\$UPDSEC

\$UPDSEC—Update Section File on Disk

The Update Section File on Disk service writes all modified pages in an active private or global section back into the section file on disk. One or more I/O requests are queued, based on the number of pages that have been modified.

FORMAT	SY\$UPDSEC <i>inadr</i> , <i>[retadr]</i> , <i>[acmode]</i> , <i>[updflg]</i> , <i>[efn]</i> , <i>[iosb]</i> , <i>[astadr]</i> , <i>[astprm]</i>
---------------	--

RETURNS	VMS Usage: cond_value type: longword (unsigned) access: write only mechanism: by value
----------------	---

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS *inadr*

VMS Usage: **address_range**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference—array reference or descriptor**

Starting and ending virtual addresses of the pages that are to be written to the section file if they have been modified. The *inadr* is the address of a two-longword array containing, in order, the starting and the ending process virtual addresses. Only the virtual page number portion of each virtual address is used; the low-order 9 bits are ignored.

The \$UPDSEC service scans pages starting at the address contained in the first longword specified by *inadr* and ending at the address contained in the second longword. Within this range, \$UPDSEC locates read/write pages that have been modified and writes (contiguously if possible) them to the section file on disk. Unmodified pages are also written to disk if they share the same cluster with modified pages.

If the starting and ending virtual addresses are the same, a single page is written to the section file if the page has been modified.

The address specified by the second longword may be smaller than the address specified by the first longword.

retadr

VMS Usage: **address_range**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference—array reference or descriptor**

Addresses of the first and last pages that were actually queued for writing, in the first \$QIO request, back to the section file on disk. The *retadr* argument is the address of a two-longword array containing, in order, the address of the first page and the address of the last page.

System Service Descriptions

\$UPDSEC

If \$UPDSEC returns an error condition value in R0, each longword specified by **retadr** will contain the value -1. In this case, an event flag is not set, no AST is delivered, and the I/O status block is not written to.

acmode

VMS Usage: **access_mode**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Access mode on behalf of which the service is performed. The **acmode** argument is a longword containing the access mode. The \$PSLDEF macro defines the symbols for the four access modes.

The most privileged access mode used is the access mode of the caller. A page cannot be written to disk unless the access mode used by \$UPDSEC is equal to or more privileged than the access mode of the owner of the page to be written.

updfg

VMS Usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Update specifier for read/write global sections. The **updfg** argument is a longword value. The value 0 (the default) specifies that all read/write pages in the global section are to be written to the section file on disk, regardless of whether or not they have been modified. The value 1 specifies that (1) the caller is the only process actually writing the global section, and (2) only those pages that were actually modified by the caller are to be written to the section file on disk.

efn

VMS Usage: **ef_number**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Event flag to be set when the section file on disk is actually updated. The **efn** argument is a longword specifying the number of the event flag.

If **efn** is not specified, event flag 0 is used.

When \$UPDSEC is invoked, the specified event flag or event flag 0 is cleared; when the update operation has completed, the event flag is set.

iosb

VMS Usage: **io_status_block**
type: **quadword (unsigned)**
access: **write only**
mechanism: **by reference**

I/O status block that is to receive the final completion status of the updating operation. The **iosb** argument is the address of the quadword I/O status block.

System Service Descriptions

\$UPDSEC

When \$UPDSEC is invoked, the I/O status block is cleared. Once the update operation has completed, that is, when all I/O to the disk has completed, the I/O status block will be written as follows:

- The first word contains the condition value returned by \$QIO, indicating the final completion status.
- The first bit in the second word will be set only if an error occurred during the I/O operation and the error was a hardware write error.
- The second longword contains the virtual address of the first page that was not written.

Though this argument is optional, DIGITAL strongly recommends that you specify it for the following reasons.

- If you are using an event flag to signal the completion of the service, you can test the I/O status block for a condition value to be sure that the event flag was not set by an event other than service completion.
- If you are using the \$SYNCH service to synchronize completion of the service, the I/O status block is a required argument for \$SYNCH.
- The condition value returned in R0 and the condition value returned in the I/O status block provide information about different aspects of the call to the \$UPDSEC service. The condition value returned in R0 gives you information about the success or failure of the service call itself; the condition value returned in the I/O status block gives you information about the success or failure of the service operation. Therefore, to accurately assess the success or failure of the call to \$UPDSEC, you must check the condition values returned in both R0 and the I/O status block.

astadr

VMS Usage: **ast_procedure**

type: **procedure entry mask**

access: **call without stack unwinding**

mechanism: **by reference—procedure reference or descriptor**

AST routine to be executed when the section file has been updated. The **astadr** argument is the address of the entry mask of this routine.

If **astadr** is specified, the AST routine executes at the access mode from which the section file update was requested.

astprm

VMS Usage: **user_arg**

type: **longword (unsigned)**

access: **read only**

mechanism: **by value**

AST parameter to be passed to the AST routine. The **astprm** argument is this longword parameter.

System Service Descriptions

\$UPDSEC

DESCRIPTION

The \$UPDSEC service uses the calling process's direct I/O limit (DIRIO) quota in queuing the I/O request and uses the calling process's AST limit (ASTLM) quota if the **astadr** argument is specified.

Proper use of this service requires the caller to synchronize completion of the update request. This is done by first checking the condition value returned in R0 by \$UPDSEC. If **SS\$_NOTMODIFIED** is returned, the caller can continue. If **SS\$_NORMAL** is returned, the caller should wait for the I/O to complete and then check the first word of the I/O status block for the final completion status. The Synchronize (\$SYNCH) service can be used to determine whether the I/O operation has actually completed.

For a global section located in memory shared by multiple processors, only processes running on the processor that created the section can specify that global section in a call to the \$UPDSEC service. Processes on another processor that attempt to update the section file will receive an error condition value indicating that the request was not performed.

CONDITION VALUES RETURNED

SS\$_NORMAL	Service successfully completed. One or more I/O requests were queued.
SS\$_NOTMODIFIED	Service successfully completed. No pages in the input address range were section pages that had been modified. No I/O requests were queued.
SS\$_ACCVIO	The input address array cannot be read by the caller, or the output address array cannot be written by the caller.
SS\$_EXQUOTA	The process has exceeded its AST limit quota.
SS\$_ILLEFC	An illegal event flag number was specified.
SS\$_IVSECFLG	An invalid flag was specified.
SS\$_NOTCREATOR	The section is in memory shared by multiple processors and was created by a process on another processor.
SS\$_NOPRIV	A page in the specified range is in the system address space.
SS\$_PAGOWNVIO	A page in the specified range is owned by an access mode more privileged than the access mode of the caller.
SS\$_SHMNOTCNCT	The section is specified as being in memory shared by multiple processors, but this shared memory is not known to the system.
SS\$_UNASCEFC	The process is not associated with the cluster containing the specified event flag.

System Service Descriptions

\$UPDSECW

\$UPDSECW—Update Section File on Disk and Wait

The Update Section File on Disk and Wait service writes all modified pages in an active private or global section back into the section file on disk. One or more I/O requests are queued, based on the number of pages that have been modified.

The \$UPDSECW service completes synchronously; that is, it returns to the caller after writing all updated pages.

For asynchronous completion, use the Update Section File on Disk (\$UPDSEC) service; \$UPDSEC returns to the caller after queuing the update request, without waiting for the pages to be updated.

In all other respects, \$UPDSECW is identical to \$UPDSEC. Refer to the documentation of \$UPDSEC for all other information about the \$UPDSECW service.

For additional information about system service completion, refer to the Synchronize (\$SYNCH) service and to Section 2.5.

FORMAT	SYSS\$UPDSECW <i>inadr</i> [<i>,retadr</i>] [<i>,acmode</i>] [<i>,updfld</i>] [<i>,efn</i>] [<i>,iosb</i>] [<i>,astadr</i>] [<i>,astprm</i>]
---------------	--

\$WAITFR—Wait for Single Event Flag

The Wait for Single Event Flag service tests a specific event flag and returns immediately if the flag is set. Otherwise, the process is placed in a wait state until the event flag is set.

FORMAT **SYSS\$WAITFR** *efn*

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENT *efn*

VMS Usage: **ef_number**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Number of the event flag for which to wait. The *efn* argument is a longword containing this number.

DESCRIPTION

The wait state caused by this service can be interrupted by an asynchronous system trap (AST) if (1) the access mode at which the AST executes is equal to or more privileged than the access mode from which the \$WAITFR service was issued and (2) the process is enabled for ASTs at that access mode.

When a wait state is interrupted by an AST and after the AST service routine completes execution, VAX/VMS repeats the \$WAITFR request on behalf of the process. At this point, if the event flag has been set, the process resumes execution.

**CONDITION
VALUES
RETURNED**

SS\$_NORMAL
SS\$_ILLEFC
SS\$_UNASEFC

Service successfully completed.
An illegal event flag number was specified.
The process is not associated with the cluster containing the specified event flag.

System Service Descriptions

\$WAKE

\$WAKE—Wake Process from Hibernation

The Wake Process from Hibernation service activates a process that has placed itself in a state of hibernation with the Hibernate (\$HIBER) service.

FORMAT **SY\$WAKE** [*pidadr*],[*prcnam*]

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS

pidadr

VMS Usage: **process_id**
type: **longword (unsigned)**
access: **modify**
mechanism: **by reference**

Process identification (PID) of the process to be awakened. The *pidadr* argument is the address of a longword containing the PID.

prcnam

VMS Usage: **process_name**
type: **character-coded text string**
access: **read only**
mechanism: **by descriptor—fixed length string descriptor**

Process name of the process to be awakened. The *prcnam* argument is the address of a character string descriptor pointing to a 1- to 15-character process name string.

The process name is implicitly qualified by the UIC group number of the calling process. For this reason, the *prcnam* argument can only be used if the process to be awakened is in the same UIC group as the calling process. To awaken a process in another UIC group, the *pidadr* argument must be specified.

If neither the *pidadr* nor the *prcnam* arguments are specified, the wake request is issued for the calling process.

DESCRIPTION Depending on the operation, use of \$WAKE may require the calling process to have one of the following privileges:

- GROUP privilege is required to wake another process in the same group, unless the process has the same UIC as the calling process.
- WORLD privilege is required to wake any other process in the system.

System Service Descriptions

\$WAKE

If one or more wake requests are issued for a process that is not currently hibernating, a subsequent hibernate request completes immediately, that is, the process does not hibernate. No count is maintained of outstanding wake-up requests.

A hibernating process can also be awakened with the Schedule Wakeup (\$SCHDWK) service.

CONDITION VALUES RETURNED

SS\$_NORMAL

Service successfully completed.

SS\$_ACCVIO

The process name string or string descriptor cannot be read by the caller, or the process identification cannot be written by the caller.

SS\$_IVLOGNAM

The specified process name string has a length of 0 or has more than 15 characters.

SS\$_NONEXPR

Warning. The specified process does not exist, or an invalid process identification was specified.

SS\$_NOPRIV

The process does not have the privilege to wake the specified process.

System Service Descriptions

\$WFLAND

\$WFLAND—Wait for Logical AND of Event Flags

The Wait for Logical AND of Event Flags service allows a process to specify a set of event flags for which it wishes to wait. The process is put in a wait state until all specified event flags are set, at which time \$WFLAND returns to the caller and execution resumes.

FORMAT **SYS\$WFLAND** *efn,mask*

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS *efn*

VMS Usage: **ef_number**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Number of any event flag within the event flag cluster that is to be used. The *efn* argument is a longword containing this number. Specifying the number of an event flag within the cluster serves to identify the event flag cluster.

There are two local event flag clusters: cluster 0 and cluster 1. Cluster 0 contains event flag numbers 0 to 31, and cluster 1 contains event flag numbers 32 to 63.

There are two common event flag clusters: cluster 2 and cluster 3. Cluster 2 contains event flag numbers 64 to 95, and cluster 3 contains event flag numbers 96 to 127.

mask

VMS Usage: **mask_longword**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Event flags for which the process is to wait. The **mask** argument is a longword bit vector wherein a bit, when set, selects the corresponding event flag for which to wait.

System Service Descriptions

\$WFLAND

DESCRIPTION

The wait state caused by this service can be interrupted by an asynchronous system trap (AST) if (1) the access mode at which the AST executes is equal to or more privileged than the access mode from which the \$WAITFR service was issued and (2) the process is enabled for ASTs at that access mode.

When a wait state is interrupted by an AST and after the AST service routine completes execution, VAX/VMS repeats the \$WFLAND request on behalf of the process. At this point, if all the specified event flags have been set, the process resumes execution.

CONDITION VALUES RETURNED

SS\$_NORMAL

Service successfully completed.

SS\$_ILLEFC

An illegal event flag number was specified.

SS\$_UNASEFC

The process is not associated with the cluster containing the specified event flag.

System Service Descriptions

\$WFLOR

\$WFLOR—Wait for Logical OR of Event Flags

The Wait for Logical OR of Event Flags service allows a process to specify a set of event flags for which it wishes to wait. The process is put in a wait state until any one of the specified event flags is set, at which time \$WFLOR returns to the caller and execution resumes.

FORMAT

SYS\$WFLOR *efn,mask*

RETURNS

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

ARGUMENTS *efn*

VMS Usage: **ef_number**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Number of any event flag within the event flag cluster that is to be used. The **efn** argument is a longword containing this number. Specifying the number of an event flag within the cluster serves to identify the event flag cluster.

There are two local event flag clusters: cluster 0 and cluster 1. Cluster 0 contains event flag numbers 0 to 31, and cluster 1 contains event flag numbers 32 to 63.

There are two common event flag clusters: cluster 2 and cluster 3. Cluster 2 contains event flag numbers 64 to 95, and cluster 3 contains event flag numbers 96 to 127.

mask

VMS Usage: **mask_longword**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Event flags for which the process is to wait. The **mask** argument is a longword bit vector wherein a bit, when set, selects the corresponding event flag for which to wait.

System Service Descriptions

\$WFLOr

DESCRIPTION

The wait state caused by this service can be interrupted by an asynchronous system trap (AST) if (1) the access mode at which the AST executes is equal to or more privileged than the access mode from which the \$WFLOr service was issued and (2) the process is enabled for ASTs at that access mode.

When a wait state is interrupted by an AST and after the AST service routine completes execution, VAX/VMS repeats the \$WFLOr request on behalf of the process. At this point, if all the specified event flags have been set, the process resumes execution.

CONDITION VALUES RETURNED

SS\$_NORMAL

Service successfully completed.

SS\$_ILLEFC

An illegal event flag number was specified.

SS\$_UNASEFC

The process is not associated with the cluster containing the specified event flag.



A User-Written System Services

Note: The original version of this material appeared in the Version 3.0 *VAX/VMS Real-Time User's Guide*, Chapter 6.

A user-written system service is a shareable image containing one or more routines that nonprivileged users can call to perform privileged functions. The creator of the user-written system service codes, compiles or assembles, links, and installs the routine; other users can then call this routine in their programs using the standard CALL interface, provided they have linked their object module(s) with the user-written system service. User-written system services thus provide a vehicle for users to write and use their own system services.

Because user-written system services can be written for any purpose, their use is not limited to real-time applications. However, user-written system services can provide real-time users with a suitable vehicle for special-purpose routines that nonprivileged processes in applications can use.

A.1 Coding the User-Written System Service

The following requirements must be met in coding a user-written system service:

- It must contain a special change-mode vector identifying a kernel-mode and/or executive-mode dispatcher.
- Its entry point must be followed by a CHMK or CHME instruction with a negative operand.
- Any kernel-mode or executive-mode dispatcher pointed to in the change-mode vector must validate the CHMK or CHME operand, and must be followed by one or more routines that perform the desired function(s).
- The user-written system service (or each routine in it) must enable any necessary user privileges and disable them when they are no longer needed. The Set Privileges (\$SETPRV) system service is used to enable and disable user privileges.

Each of the preceding considerations is discussed in the following sections.

A.1.1 Change-Mode Vector

One of the program sections in a user-written system service must start with a change-mode vector. The purpose of this vector is to point (by means of self-relative offsets) to the start of the kernel-mode or executive-mode dispatch routine within the user-written system service.

The program section containing the change-mode vector must be assigned the VEC attribute. (See the *VAX MACRO and Instruction Set Reference Manual* or the *VAX/VMS Linker Utility Reference Manual* for a discussion of program section attributes.)

The change-mode vector must have the format shown below. The offsets from the base of the vector to specific items are expressed by symbols starting with PLV\$L_. These symbols are defined by the \$PLVDEF macro and are contained in SYS\$LIBRARY:STARLET.MLB.

Symbols Defined by \$PLVDEF Macro

```
Vector Type Code  PLV$L_TYPE  
(PLV$C_TYP_CM0D)  
System Version Number  PLV$L_VERSION  
(SYS$K_VERSION)  
Kernel Mode Dispatcher Offset  PLV$L_KERNEL  
Exec Mode Entry Offset  PLV$L_EXEC  
User Rundown Service Offset  PLV$L_USRUNDWN  
Reserved  
RMS Dispatcher Offset  PLV$L_RMS  
Address Check  PLV$L_CHECK
```

The significant offsets in the change-mode vector and their contents are as follows:

- PLV\$L_TYPE—Contains the type code PLV\$C_TYP_CM0D, identifying this as a change-mode vector.
- PLV\$L_VERSION—Contains the system version number (expressed by the value SYS\$K_VERSION). When the user-written system service is linked, the linker inserts the value of SYS\$K_VERSION into this location. Before the user-written system service is used at run time, the VAX/VMS image activator compares this value with the current version number of SYS.EXE; and if the two do not match, the user-written system service is not used and an error status is returned.
- PLV\$L_KERNEL—Contains a self-relative pointer to the user-supplied kernel-mode dispatcher. ("Self-relative" means relative to the start of the longword field.) A zero value indicates there is no kernel-mode dispatcher.
- PLV\$L_EXEC—Contains a self-relative pointer to the user-supplied executive-mode dispatcher. A zero value indicates there is no executive-mode dispatcher.
- PLV\$L_USRUNDWN—Contains a self-relative pointer to the user-supplied rundown routine. This offset is optional. This routine is intended to be used for image-specific cleanup and resource deallocation. When the image linked against the user-written system service is run down by the system, this run-time routine is invoked. Unlike exit handlers, it is always called when a process or image exits. (This routine is called with a JSB instruction and returns with a RSB instruction in kernel mode, at IPL 0.) For information about exit handlers, see Section 8.6.3.
- PLV\$L_RMS—Contains a self-relative pointer to the dispatcher for VAX RMS services. A zero value indicates there is no user-supplied VAX RMS dispatcher. Only one user-written system service should specify the VAX RMS vector, because only the last value will be used. This field is intended for use only by DIGITAL.
- PLV\$L_CHECK—Contains a value to verify that a user-written system service that is not position-independent is located at the proper virtual address. If the image is position-independent, this field should contain zero. If the image is not position-independent, this field should contain its own address.

A.1.2 Entry Point to the User-Written System Service

The entry point of a user-written system service must be an entry mask followed by a CHMK (Change Mode to Kernel) or CHME (Change Mode to Executive) instruction, depending on whether you want control transferred to a kernel-mode or executive-mode dispatcher (specified in the vector). The operand of the CHMK or CHME instruction must be a negative value, because positive values are reserved for calling system services supplied by DIGITAL.

A.1.3 Kernel-Mode or Executive-Mode Dispatcher

The kernel-mode or executive-mode code that you write must

- Validate the CHMK or CHME operand, handling any invalid operands.
- Transfer control to the appropriate coding segment if the user-written system service contains functionally separate coding segments. The CASE instruction in VAX MACRO or a computed GOTO-type statement in a high-level language provides a convenient mechanism for determining where to transfer control.
- Precede the coding segments performing the functions the user-written system service was designed to perform.

A.1.4 Enabling and Disabling User Privileges

A user-written system service must enable any privileges that it needs but that the nonprivileged user of the user-written system service lacks. The user-written system service must also disable any such privileges before the nonprivileged user receives control again. To enable or disable a set of privileges, use the Set Privileges (\$SETPRV) system service. The following example shows the operator (OPER) and physical I/O (PHY_IO) privileges being enabled.

```
PRVMSK: .LONG <1@PRV$V_OPER>!<1@PRV$V_PHY_IO> ;OPER AND PHY_IO
        .LONG 0 ;QUADWORD MASK REQUIRED. NO BITS SET IN
                ;HIGH-ORDER LONGWORD FOR THESE PRIVILEGES.
```

```
$SETPRV_S ENBFLG=#1,- ;1=enable, 0=disable
          PRVADR=PRVMSK ;Identifies the privileges
```

Any code executing in executive or kernel mode is granted an implicit SETPRV privilege.

A.2 Linking the User-Written System Service

The following conventions apply when you link (that is, create) a user-written system service:

- Use the /SHAREABLE command qualifier to identify the image to be created as shareable.
- Use the /PROTECT command qualifier or the PROTECT= option to identify the entire image or specific clusters, respectively, as protected against user-mode or supervisor-mode write access.
- Define the user-written system service's entry point as a universal symbol, using the UNIVERSAL= option.

A.2.1 Specifying Protection for the Image or Clusters

The VAX/VMS Linker allows you to protect all or part of a user-written system service from write access by code executing in user or supervisor mode. The /PROTECT command qualifier causes all image sections to be so protected. The PROTECT= option in a linker options file permits you to specify protection for individual clusters, thus allowing user-written system services to contain parts into which the nonprivileged user can write.

The linker option takes the form PROTECT=YES or PROTECT=NO and precedes the specifications for clusters that are to be protected or unprotected, respectively. The following example shows the linker options file entries to designate clusters A, B, and D as protected, and cluster C as unprotected.

```
PROTECT=YES
CLUSTER=A,,,MODULE1,MODULE2
CLUSTER=B,,,MODULE3,MODULE4,MODULE5
PROTECT=NO
CLUSTER=C,,,MODULE6,MODULE7
PROTECT=YES
CLUSTER=D,,,MODULE8,MODULE9
```

The *VAX/VMS Linker Utility Reference Manual* discusses linker options files and explains each available option.

A.3 Installing the User-Written System Service

To make a user-written system service usable by nonprivileged programs, you must install it as a protected permanent global section. The following procedure is recommended:

- 1 Move the user-written system service to a protected directory, such as SYS\$SHARE.
- 2 Run the Install Utility, specifying the /PROTECT, /OPEN, and /SHARED qualifiers. You can also specify the /HEADER_RESIDENT qualifier. The following entry could be used to install a user-written system service whose image name is USS:

```
* RUN SYS$SYSTEM:INSTALL
INSTALL>SYS$SHARE:USS/PROTECT/OPEN/SHARED/HEADER_RES
```

The Install Utility is discussed in the *VAX/VMS Install Utility Reference Manual*.

A.4 Using the User-Written System Service

To the nonprivileged user of a user-written system service there is no difference between using it and using an ordinary shareable image. To use a user-written system service, the user must

- Call the user-written system service.
- Link the user-written system service into the executable image being created. Note: If the user-written system service was installed as writeable, you cannot link it into an executable image. You must link an uninstalled copy of the writeable user-written system service into the executable image.

A.5 Program Listings

Refer to SYS\$EXAMPLES:USSDISP.MAR for listings of modules in a user-written system service and of a module that calls the user-written system service.



B Using Shared Memory

Note: The original version of this material appeared in the Version 3.0 *VAX/VMS Real-Time User's Guide*, Chapter 5.

The MA780 is a multiport memory unit that can be attached to VAX-11/780 processors. Each VAX-11/780 processor can support up to two MA780s. Each MA780 has four ports, thereby allowing up to four VAX-11/780 processors to be attached to it.

Using one or more multiport memory units, an application can consist of multiple processes running on different VAX-11/780 processors. Regardless of the processor on which they are running, these processes can communicate the completion of an event, send messages, and share common data and code by means of the shared memory.

B.1 Preparing Multiport Memory for Use

Before an application using multiport memory can execute under VAX/VMS, the system manager must activate the VAX/VMS operating system in the processors connected to the multiport memory unit and initialize that memory. The *VAX/VMS System Manager's Reference Manual* explains the system management responsibilities associated with a multiport memory unit; the present section summarizes the system management functions for the benefit of the application programmer.

First, the system manager activates the VAX/VMS operating system in a VAX-11/780 and initializes the multiport memory unit. These actions cause the following to occur:

- The uninitialized shared memory is connected to the VAX/VMS running in the processor.
- A name is defined that all processes running in all processors can use to refer to the shared memory (see Section B.3)
- Limits are set for the following resources in this multiport memory unit:
 - Common event flag clusters: the total number that can be created, and the number that can be created by processes running on this processor
 - Mailboxes: the total number that can be created, and the number that can be created by processes running on this processor
 - Global sections: the total number that can be created, and the number that can be created by processes running on this processor

Then the system manager activates the VAX/VMS operating system in the other processors connected to the multiport memory unit. The system manager then connects the initialized shared memory to the VAX/VMS system running in each of these processors and sets limits for the number of common event flag clusters, global sections, and mailboxes that processes on each processor can create in the multiport memory.

The system manager can also install global sections in shared memory just as they are installed in local memory. The Install Utility can be used to create shared memory global sections for known files. Once the global sections are installed, a process running in any processor connected to the multiport memory can map to the section, if the process has the appropriate privilege. The process can gain access to the global section either by using a logical name defined by the system manager or by using the section name specified when the global section was created. In the latter case, the section name must be unique on the processor running the process that is attempting to access the global section.

B.2 Privileges Required for Shared Memory Use

To use facilities in memory shared by multiple processors, you must have all of the user privileges required to use the equivalent facility in local memory. For example, to create a permanent global section, you must have the PRMGBL privilege, and to create a temporary or permanent mailbox, you must have the TMPMBX or PRMMBX privilege, respectively.

In addition to any other required privileges, you must have the SHMEM privilege to create or delete a common event flag cluster, mailbox, or global section in memory shared by multiple processors. However, you do not need the SHMEM privilege to use an existing cluster, mailbox, or global section in multiport memory.

B.3 Naming Facilities in Shared Memory

To allow access to facilities in memory shared by multiple processors, the system manager and application programmers define names that application programs use to refer to individual shared memory units. During system installation, the system manager defines the name that processes on that particular processor use to refer to the shared memory itself. Application programs define the names that they use to refer to common event flag clusters, global sections, and mailboxes located in the shared memory.

By convention, facilities in shared memory have a name string in the following format:

memory-name:facility-name

memory-name	Name assigned by the system manager during system installation to the shared memory containing the facility. VAX/VMS requires the memory name when you specify a common event flag cluster or mailbox. The colon is recognized as a delimiter separating the two parts of the name string. The name must contain 43 or fewer characters, and can consist only of alphabetic characters, numeric characters, the dollar sign (\$), and the underscore (_).
facility-name	Logical name assigned to the event flag cluster, global section, or mailbox. The name must contain 43 or fewer characters, and can consist only of alphabetic characters, numeric characters, the dollar sign (\$), and the underscore (_).

Examples of facility names are:

SHRMEM:GS_DATA	Identifies the global section GS_DATA in the shared memory named SHRMEM
SHRMEM:MAILBX	Identifies the mailbox MAILBX in the same shared memory

B.4 Assigning Logical Names and Logical Name Translation

You can define a logical name for a shared memory facility with the DEFINE or ASSIGN command or the Create Logical Name (\$CRELNM) system service. Application programs can then refer to the facility using the logical name; for example, a process can invoke the Create Mailbox and Assign Channel (\$CREMBX) system service specifying the logical name for an existing mailbox to which a channel is to be assigned.

When translating a logical name for a shared memory facility, the VAX/VMS operating system uses a slightly different approach from that used for other logical names. The purpose of this approach is to allow programmers to specify either the complete name (memory name and facility name) or a logical name that the system will translate to the complete name. If you define logical names properly, a program that uses a given facility in local memory can be run without change to use the facility in shared memory.

Whenever VAX/VMS encounters the name of a common event flag cluster, mailbox, or global section, it performs the following special logical name translation sequence:

- 1 Inserts one of the following prefixes to the name (or to the part of the name before the colon if a colon is present):
 - CEF\$ for common event flag clusters
 - MBX\$ for mailboxes
 - GBL\$ for global sections
- 2 Subjects the resultant string to logical name translation. If translation does not succeed (that is, the original name did not use a logical name), passes the original name string to the system service. If translation does succeed, goes to step 3.
- 3 Appends the part of the original string after the colon (if any) to the translated name.
- 4 Repeats steps 1 to 3 (up to a number of times determined by the system, if necessary) until logical name translation fails. When translation fails, passes the string to the system service.

For example, assume that you have made the following logical name assignment:

```
* DEFINE MBX$CHKPNT SHRMEM$1:CHKPNT
```

Assume also that your program refers to the mailbox name as CHKPNT in a system service argument. The following logical name translation takes place:

- 1 MBX\$ is prefixed to CHKPNT.
- 2 MBX\$CHKPNT is translated to SHRMEM\$1:CHKPNT.
- 3 No further translation is successful; therefore, the string SHRMEM\$1:CHKPNT is passed to the system service.

The logical name definition in the preceding example allows a program that used a mailbox named CHKPNT in local memory to run using the mailbox in shared memory, without being recompiled or relinked.

Note that if a process creates one or more subprocesses and they use a mailbox or common event flag cluster in shared memory, the creator should place the logical name in the job or group logical name table (for example, specify the /JOB or /GROUP qualifier with the DEFINE command). If the name is defined in the process logical name table (the default), the subprocesses will not receive the correct equivalence name, because each subprocess has its own process logical name table.

There are two exceptions to the logical name translation method discussed in this section:

- If the facility name starts with an underscore (_), the VAX/VMS system strips the underscore and considers the resultant string to be the actual name (that is, no further translation is performed).
- If the facility is a global section with a name in the format name_nnn, VAX/VMS first strips the underscore and the digits (nnn), then translates the resultant name according to the sequence discussed in this section, and finally reappends the underscore and digits. The system uses this method with known images and shared files installed by the system manager.

B.5 How VAX/VMS Finds Facilities in Shared Memory

After the VAX/VMS system performs the logical name translation described in Section B.4, the final equivalence name must be the name of a facility in either the processor's local memory or in shared memory. If the equivalence name specifies the name of a shared memory (that is, the name is in the format memory-name:facility-name), VAX/VMS searches for the facility in the appropriate database of the specified shared memory unit.

If the equivalence name specifies a common event flag cluster or mailbox and does not specify a memory name, VAX/VMS searches through the local memory common event flag cluster database or mailbox database until it locates the specified cluster or mailbox. Absence of a memory name as part of a common event flag cluster name or mailbox name indicates that the facility is located in local memory.

If the equivalence name specifies a global section and does not specify a memory name, VAX/VMS looks for the section as follows:

- 1 First, it searches the global section tables for sections in the processor's local memory.
- 2 Then, it searches the global section tables for each initialized shared memory connected to the processor in the order in which they were connected and recognized by the processor.

The result of searching in this order is that global sections in the processor's local memory take precedence over those in shared memories. Thus, absence of a memory name as part of a global section name is not used as an indication of where the global section is located.

B.6 Using Common Event Flags in Shared Memory

Under VAX/VMS, any process can associate with up to two common event flag clusters (event flag numbers 64 through 95 and 96 through 127). These clusters can be located in shared memory or in local memory. To create and associate with a common event flag cluster in shared memory and manipulate flags in the cluster, you use the same steps as you would to associate with a common event flag cluster in local memory:

- 1 Issue the Associate Common Event Flag Cluster (\$ASCEFC) system service to create and name the cluster or to associate with an existing named cluster.
- 2 Issue any of the services that set, clear, and wait for designated event flags, as appropriate.

As with local memory clusters, the first process among cooperating processes to issue the Associate Common Event Flag Cluster (\$ASCEFC) system service causes the cluster to be created and named. Any other process calling this service and specifying the same cluster name associates with that existing cluster. VAX/VMS implicitly qualifies cluster names with the group number of the creator's UIC; therefore, other cooperating processes must belong to the same group. All of the event flag system services, with the exception of Associate Common Event Flag Cluster and Disassociate Common Event Flag Cluster, function identically regardless of whether they are used with local or shared memory clusters. The only difference with the associate and disassociate system services is that to specify a cluster in shared memory, you must provide the memory name as well as the cluster name. That is, after VAX/VMS performs logical name translation of the name argument, the cluster name must have the following format:

`memory-name:cluster-name`

Section B.3 describes the name format, and Section B.4 explains the logical name translation performed by the system. Section 4 describes all of the event flag services in detail.

B.7 Using Mailboxes in Shared Memory

The first process on each processor must use the Create Mailbox and Assign Channel (\$CREMBX) system service to create a shared memory mailbox and assign a channel to it. Any \$CREMBX system service call referring to a shared memory mailbox must specify a mailbox name that has or translates to the following format (Section C.4 explains the logical name translation procedure):

`memory-name:mailbox-name`

When the mailbox is created, the \$CREMBX system service also creates the mailbox-name portion of the name string as a logical name with an equivalence name in the format MBn. For example, if the complete name string is SHMEM:MAILBOX, the system service will create MAILBOX as a logical name with an equivalence name of, for example, MBB005.

The Assign I/O Channel (\$ASSIGN) and Deassign I/O Channel (\$DASSGN) system services require that you specify only the mailbox-name portion of a shared memory mailbox name string. Likewise, any high-level language program statements that open, close, read from, or write to a shared memory mailbox must specify only the mailbox-name portion.

The following code example shows two VAX FORTRAN programs using a shared memory mailbox. The memory name in this example is SHMEM. The programs are running in processes on separate processors.

```
PROGRAM ONE
INTEGER*4 SYS$CREMBX,STATUS,CHAN
STATUS = SYS$CREMBX(,CHAN,,,,,'SHMEM:MAILBOX')
IF (.NOT. STATUS) CALL LIB$STOP(%VAL(STATUS))
C-- Open the mailbox using the mailbox-name; write a message.
OPEN (UNIT=1,NAME='MAILBOX',STATUS='NEW')
WRITE (1,*) MESSAGE
.
.
.
END

PROGRAM TWO
INTEGER*4 SYS$CREMBX,STATUS,CHAN
STATUS = SYS$CREMBX(,CHAN,,,,,'SHMEM:MAILBOX')
IF (.NOT. STATUS) CALL LIB$STOP(%VAL(STATUS))
C-- Open the mailbox using the mailbox-name; read the message.
OPEN (UNIT=1,NAME='MAILBOX',STATUS='OLD')
READ (1,*) MESSAGE
.
.
.
END
```

A mailbox in shared memory cannot be used as a process termination mailbox. Note that because the processes run on different processors, each must issue a \$CREMBX system service request.

Section 7.18 discusses mailboxes and related system services in detail.

B.8 Using Global Sections in Shared Memory

Under VAX/VMS, processes can map global sections located in local memory or in shared memory. A global section in shared memory can be mapped to an image file or a data file, just like a global section in local memory. To create a global section in shared memory, you perform the same steps as you would to create a global section in local memory:

- 1 Using VAX RMS, open the file to be mapped.
- 2 Issue the Create and Map Section (\$CRMPSC) system service.

The file to be mapped must reside on a disk device attached to the local processor. Once the section is created, however, processes on all processors attached to the shared memory can map the section. To map to an existing global section in shared memory, you issue a Map Global Section (\$MGBLSC) system service specifying the name of the section. Once the section is mapped, processes gain access to shared memory global sections in the same manner as they do to local memory sections. VAX/VMS thus makes use of the shared memory unit transparent to the process.

VAX/VMS treats the pages of a global section in shared memory differently from pages in local memory. When a process creates a shared-memory global section, VAX/VMS brings all of the pages of the mapped image or data file into memory. Any process mapped to that global section can gain access to those pages without incurring a page fault because the pages are already in physical memory. Unlike process pages in local memory, global section pages

in shared memory are not included in the working sets of the processes that map the section.

Because no paging occurs, VAX/VMS never writes the contents of shared memory global section pages back to their disk file. For read/write global sections in which you want to maintain an updated file while the application executes, you must issue an Update Section File on Disk (\$UPDSEC) system service. The process issuing the update request must execute on the same processor as the process that created the global section. You can update the disk file periodically during execution of the application as a checkpoint precaution. The disk file is automatically updated when the section is deleted.

Each process that has mapped a global section in shared memory can unmap the section in either of the following ways:

- Issue a Delete Virtual Address Space (\$DELTVA) system service to delete the process's virtual address space that maps the section.
- Terminate the current image, thereby causing VAX/VMS to unmap the process from the section automatically.

Deleting a global section in shared memory requires an explicit deletion request, because all global sections in shared memory must be permanent sections. The deletion request can be either a Delete Global Section (\$DGBLSC) system service issued by the application or a deletion request issued by the system manager using the Install Utility. In either case, VAX/VMS does not perform the actual deletion until all processes that have mapped the section unmap it.

When a process requests deletion of a shared memory global section page, VAX/VMS will wait until there is no direct I/O outstanding for the process before deleting the page. This is because no reference count is maintained for shared memory global section pages. (For example, VAX/VMS cannot determine whether outstanding direct I/O is for the shared memory global section page or not.) Applications using devices that have direct I/O perpetually outstanding, such as the DR32, must not delete shared memory global section pages because this will cause the process to hang in the MWAIT state (unless the applications cancel the outstanding I/O request first).

B.8.1 Removing Shared Memory Global Sections

A shared memory global section can only be deleted by the creator processor.

If you rebootstrap a processor and reconnect it to an MA780 without reinitializing the MA780, the System Generation Utility (SYSGEN) does cleanup for the processor. This cleanup causes all global sections created by processes running on this processor to be marked as having no creator processor. (The data structures that allow the data in the global section pages to be written back into the disk file no longer exist.)

Without a creator processor, you must do the following before you attempt to delete shared memory global sections:

- 1 Reboot all processors
- 2 Reinitialize the MA780

Section 11.6 provides information on the use of the VAX/VMS system services used with global sections, that is, memory management system services. Section B.8.2 of this manual provides information specifically related to creating and mapping a global section in shared memory. The \$CRMPSC, \$MGBLSC, \$DGBLSC, and \$UPDSEC system services are the only memory management system services for which the shared memory has any direct implications.

B.8.2 Create and Map Section System Service

The Create and Map Section system service has the following general formats when issued to create and/or map a global section in shared memory.

VAX MACRO Format

```
$CRMPSC [inadr], [retadr], [acmode], [flags], gadnam
        , [ident], [relpag], [chan], [pagcnt], [vbn], [prot]
```

High-Level Language Format

```
SYS$CRMPSC[inadr], [retadr], [acmode], [flags], gadnam
        , [ident], [relpag], [chan], [pagcnt], [vbn], [prot]
```

With the exception of the FLAGS, GSDNAM, and PFC arguments, the arguments of this service are not affected by MA780 considerations.

Arguments

flags

Mask defining the section type and characteristics. Of the flags defined, the following two must be set.

Flag	Meaning
SEC\$_M_GBL	Global section
SEC\$_M_PERM	Permanent section

That is, sections in shared memory must be permanent global sections.

If appropriate, the following flags also can be set.

Flag	Meaning	Default
SEC\$_M_DZRO	Pages are demand-zero pages	Pages are not zeroed when copied
SEC\$_M_WRT	Read/write section	Read-only
SEC\$_M_SYSGBL	System global section	Group global section
SEC\$_M_EXPREG	Map section into the first free range of virtual addresses large enough to hold the section	Map section according to the INADR argument

Neither SEC\$_M_CRF (copy-on-reference) nor SEC\$_M_PFNMAP (page frame number mapping) can be set when using the Create and Map Section system service to create global sections in shared memory. If SEC\$_M_CRF is set, VAX/VMS places the global section in local memory.

gsdnam

Address of a character string descriptor pointing to the text name string for the global section. This argument is required for creating sections in shared memory.

The string can be either the name of a global section or the logical name of a global section. VAX/VMS performs logical name translation as described in Section B.4.

VAX/VMS implicitly qualifies global section names with an identification. For group global sections, the section name is also implicitly qualified by the group number of the process creating the global section.

pfc

Page fault cluster size for local memory sections. This argument is ignored for global sections in shared memory, because VAX/VMS reads the file into memory when it creates the section and does not allow paging for sections in shared memory.

Index

A

- Absolute time • 9-2
 - as input to SYS\$BINTIM • SYS-24
 - converting to numeric • SYS-321
 - in system format • 9-3
- Access
 - logical I/O • 7-8
 - physical I/O • 7-8
- Access mode • 2-3
 - changing to executive • SYS-55
 - changing to kernel • SYS-57
 - effect on AST delivery • 5-5
 - specifying • 2-3
 - with AST • 5-2
- Accounting message
 - format • SYS-86
- ACE (access control entry)
 - alarm • 3-17
 - application • 3-18
 - creating • 3-16, 3-23
 - default protection • 3-19
 - identifier • 3-20
 - maintaining • 3-16, 3-23
 - translating • 3-16, 3-22
 - types of • 3-17
- ACL (access control list) • 3-2
- Alarm ACE • 3-17
 - format of • 3-17
 - purpose of • 3-17
- Allocation class • SYS-195
- Application ACE • 3-18
 - format of • 3-18
 - purpose of • 3-18
- Argument
 - characteristics of
 - passing mechanism • 1-6
 - mechanism array • 10-10
 - signal array • 10-9
 - specifying • 2-8
 - VMS usage • 1-6
- Argument list • 2-4
 - creating • 2-8
 - for AST service routine • 5-4
 - for condition handler • 10-9
 - for system services • 2-4

Array

- mechanism • 10-10
 - signal • 10-9
 - virtual address • 11-5
- ASCII string
 - converting to binary • SYS-24
- ASCII time • 9-7
- ASSIGN command • 6-2
- AST (asynchronous system trap)
 - access mode • 5-2
 - blocking • 12-9, 12-14
 - declaring • 5-3, SYS-111
 - delivery • 5-5
 - disabling • SYS-356
 - enabling • SYS-356
 - example • 5-5
 - parameter • 5-4
 - quota • 7-3
 - service routine • 5-4
 - setting for power recovery • SYS-364
 - setting timer for • SYS-362
 - system service • 5-1
- ASTLM (AST limit) quota
 - effect of canceling wakeup • SYS-41
- Asynchronous system trap
 - See AST

B

- Balance set
 - swapping • 11-7
- Binary value
 - converting to ASCII string • SYS-155
- BIOLM (buffered I/O limit) quota • 7-3
- Blocking AST
 - description • 12-9
 - using • 12-14
- BYPASS privilege • 7-6
- BYTELM (buffered I/O byte count) quota • 7-3

C

- Call
 - testing for successful completion of • 2-15

Index

Call frame
 removing from stack • SYS-456
CALLG (Call Procedure with General Argument List)
 instruction
 example • 2-11
 macro • 2-10
CALLS (Call Procedure with Stack Argument List)
 instruction
 argument • 2-6
 example • 2-10
 macro • 2-10
Call stack
 removing frame from • SYS-456
 unwinding • 10-13
Change mode handler • 10-6
 declaring • SYS-113
Channel
 assigning I/O • 7-13, SYS-20
 canceling I/O • SYS-36
 deassigning • 7-18
Characteristic
 getting information about
 asynchronously • SYS-239
Clock
 setting system • 9-7
Common event flag cluster • 4-5
Compatibility mode handler • 10-6
 declaring • SYS-113
Condition
 for exception • 10-1
Condition handler
 argument list • 10-9
 course of action • 10-12
 example • 10-12
 specifying • 10-7
Condition-handling services • 10-1
Condition value • 2-14
 high-level language • 2-17
 information • 2-15
 testing • 2-15
Control region • 11-2
 adding page to • SYS-153
 deleting page from • SYS-59
Convention
 for calling • 2-1

D

Date
 getting current system • 9-2
 Smithsonian base • 9-2

Index-2

Date (cont'd.)
 system format • 9-2
Deadlock detection • 12-6
Default form • SYS-412
Default logical name table
 group • 6-6
 job • 6-5
 process • 6-4
 system • 6-6
Default protection ACE • 3-19
DEFINE command • 6-2
Delta time • 9-2
 as input to SYS\$BINTIM • SYS-24
 converting to numeric • SYS-321
 example • 9-3
 in system format • 9-3
Detached process • 8-2, 8-7, SYS-88
Device
 allocating • 7-20, SYS-9
 deallocating • 7-22, SYS-107
 dual-pathed • SYS-195
 getting information about • 7-25
 asynchronously • SYS-192
 synchronously • SYS-208
 implicit allocation • 7-22
 lock name • SYS-198
 name • 7-24
 default • 7-25
 protection • 7-6
 served • SYS-202
DIOLM (direct I/O limit) quota • 7-3
Directive
 SYS\$FAO • SYS-157
Directory logical name table
 process • 6-3
 system • 6-3
Disk file
 opening • 11-9
Disk volume
 mounting • 7-22
Dispatcher
 exception • 10-7
DYNAMIC attribute • 3-4

E

Equivalence name
 defining • 6-1
 format convention • 6-9
 specifying • SYS-61
Error check • 2-16

Error logger
 sending message to • SYS-393
 Error recovery • 7-12
 Event flag
 clearing • 4-4, SYS-54
 for interprocess communication • 8-10
 getting current status • SYS-341
 number • 4-2
 setting • 4-4, SYS-357
 specifying • 4-2
 wait • 4-3
 waiting for entire set of • SYS-466
 waiting for one of set • SYS-468
 waiting for setting of • SYS-463
 Event flag cluster • 4-2
 associating with a process • SYS-12
 deleting • 4-7, SYS-136
 disassociating • 4-6, SYS-106
 getting current status • SYS-341
 in shared memory • 4-8
 number • 4-2
 specifying name for • 4-8
 Exception
 dispatcher • 10-7
 generating on system service failure • SYS-378
 multiple • 10-16
 type • 10-1
 Exception vector
 setting • SYS-358
 Execution context • 8-2
 Executive mode
 changing to • SYS-55
 Exit
 forced • 8-15
 image • 8-13
 Exit handler • 8-15
 canceling • SYS-38
 control block • SYS-115
 deleting • SYS-38
 declaring • SYS-115
 Extent
 defining section • 11-10

F

File
 getting information about
 asynchronously • SYS-239
 File specification
 parsing components of • SYS-169

File specification (cont'd.)
 searching string for • SYS-169
 Forced exit • 8-15
 Foreign device • 7-7
 Foreign volume • 7-4, 7-5, 7-7
 Form
 getting information about
 asynchronously • SYS-239
 Function code • 7-11
 Function modifier • 7-12
 IO\$_M_DATACHECK • 7-12
 IO\$_M_INHERLOG • 7-7
 IO\$_M_INHRETRY • 7-12

G

Global section • 11-11
 characteristic • 11-11
 creating • SYS-96
 defining • 11-8
 deleting • SYS-130
 for interprocess communication • 8-10
 mapping • 11-15, SYS-96, SYS-305
 name format • 11-12
 paging file • 11-16
 Granularity
 in lock • 12-2
 Group logical name table • 6-6

H

Handler
 change and compatibility mode • 10-6
 Hibernation • 8-11
 alternate method • 8-12
 and AST • 5-3
 compared with suspension • 8-11
 High-level language
 call from • 2-17
 Holder record • 3-5
 adding • 3-8
 format of • 3-5
 modifying • 3-12
 removing • 3-14
 Host • SYS-195

Index

I

- I/O channel • 7-13
 - assigning • SYS-20
 - deassigning • 7-18, SYS-109
- I/O completion
 - recommended test • 7-16
 - status • 7-17
 - synchronizing • 7-14
- I/O device
 - getting information about
 - asynchronously • SYS-192
 - synchronously • SYS-208
- I/O function
 - code • 7-11, 7-13
 - modifier • 7-12
- I/O operation
 - logical • 7-7
 - physical • 7-7
 - quotas, privileges, and protection • 7-2
 - summary of • 7-6
 - virtual • 7-7
- I/O request
 - canceling • 7-20
 - canceling on channel • SYS-36
 - queuing • 7-13
 - asynchronously • SYS-334
 - synchronously • SYS-340
- I/O service
 - synchronous version • 7-17
- I/O status block
 - in synchronization • 7-14
 - return condition value field • 7-17
- Identifier • 3-2
 - adding to rights database • 3-8
 - attributes • 3-4
 - defining • 3-2
 - determining holders of • 3-9
 - format of • 3-2, 3-3
 - general • 3-4
 - removing from rights database • 3-14
 - system-defined • 3-3
 - UIC format • 3-3
- Identifier ACE • 3-20
- Identifier name • 3-3
 - translating • 3-7
- Identifier record • 3-5
 - adding to rights database • 3-8
 - format of • 3-5
 - modifying • 3-12
 - removing from rights database • 3-14

- Identifier value
 - translating • 3-7
- IFI (internal file identifier)
 - removing • 6-10
- Image
 - exit • 8-13, SYS-152
 - for subprocess • 8-3
 - rundown activity • 8-14
- Image rundown
 - effect on logical names • 6-5
 - forcing • SYS-181
- Image section • 11-19
- Input address array • 11-4
- Internal file identifier
 - See IFI
- Interprocess
 - communication • 8-7, 8-9
 - using event flags for • 8-10
 - using global sections for • 8-10
 - using lock management services for • 8-10
 - using logical names for • 8-10
 - using mailboxes for • 8-10
 - control • 8-7

J

- Job
 - getting information about
 - asynchronously • SYS-209, SYS-239
 - synchronously • SYS-222.2
- Job controller
 - major interface
 - asynchronous • SYS-393
 - synchronous • SYS-428.8
- Job logical name table • 6-5

K

- Kernel mode
 - changing to • SYS-57

L

- Local buffer caching
 - with lock management service • 12-13
- Lock
 - choice of mode • 12-3

Lock (cont'd.)
 concept of • 12-1
 conversion • 12-6, 12-9
 deadlock detection • 12-6
 dequeuing • 12-12
 getting information about
 asynchronously • SYS-223
 synchronously • SYS-234
 level • 12-4
 mode • 12-3
 Lock database
 in a VAXcluster • SYS-232
 Lock management service
 for interprocess communication • 8-10
 Lock request
 completing • 7-14
 dequeuing • SYS-126
 queuing • 12-4
 asynchronously • SYS-138
 synchronously • SYS-148
 synchronizing • 7-14, 12-7
 Lock status block • 12-8, SYS-140
 in synchronization • 7-14
 Lock value block • SYS-140
 description • 12-11
 using • 12-13
 Logical I/O
 access checks • 7-9
 operations • 7-7
 privilege • 7-4, 7-7
 Logical name • 7-24
 attributes • 6-7
 creating • 6-11, SYS-61
 defining • 6-1
 deleting • 6-15, SYS-117
 duplicating • 6-12
 for interprocess communication • 8-10
 format convention • 6-9
 getting information about • SYS-447
 image rundown • 6-5
 multivalued • 6-2
 supersession • 6-14
 translating • 6-16, SYS-447
 Logical name system service call
 example of
 SYS\$CRELNM • 6-11
 SYS\$CRELNT • 6-15
 SYS\$DELLNM • 6-15
 SYS\$TRNLNM • 6-16
 Logical name table
 creating • 6-14, SYS-66
 default • 6-3

Logical name table (cont'd.)
 deleting • SYS-117
 directory • 6-3
 group • 6-6
 job • 6-5
 predefined logical names • 6-2
 process • 6-4
 process-private • 6-6
 quotas • 6-8
 search list • 6-10
 modifying • 6-10
 shareable • 6-6, 6-15
 system • 6-6
 types of • 6-2
 user-defined • 6-6

M

MACRO

CALLG (Call Procedure with General Argument List) instruction • 2-10
 CALLS (Call Procedure with Stack Argument List) instruction • 2-10

Macro

calling system service using • 2-9
 expansion • 2-8
 system service • 2-1, 2-5

Mailbox • 7-28

assigning channel to • SYS-72
 creating • SYS-72
 deleting
 permanent • SYS-75, SYS-120
 temporary • SYS-75
 for interprocess communication • 8-10
 for system process • 7-33
 name format • 7-31
 protection • 7-4, 7-5
 system • 7-32
 messages • 7-32
 termination • 8-19

Mechanism array argument • 10-10

Memory

locking page into • 11-7, SYS-301
 unlocking page from • SYS-452

Message

formatting and outputting • SYS-326
 obtaining text of • SYS-235
 sending to error logger • SYS-393
 sending to operator • SYS-429
 system • 2-16
 writing to terminal • SYS-27, SYS-35

Index

MOUNT privilege • 7-4
Multiple exception • 10-16

N

NARGS keyword • 2-8
Null device • 7-26
Numeric time • 9-7

O

Operator
 sending message • SYS-429
Output
 formatting character string • SYS-155
Out swap
 by suspension • 8-13

P

Page • 11-2
 copy-on-reference • 11-11
 demand-zero • 11-11
 locking into memory • 11-7, SYS-301
 locking into working set • SYS-303
 owner • 11-5
 ownership and protection • 11-5
 removing from working set • SYS-325
 setting protection • SYS-369
 unlocking from memory • SYS-452
 unlocking from working set • SYS-454
Page frame section • 11-19
Paging file section • 11-16
 global • 11-16
Parent lock • 12-10
Physical I/O
 access checks • 7-8
 operations • 7-7
 privilege • 7-4, 7-7
Physical name • 7-24
PID (process identification number) • 8-7
Power recovery
 setting AST for • SYS-364
Predefined logical name
 LNMS\$FILE_DEV • 6-11
Priority
 setting • SYS-366

Private section
 defining • 11-8
Privilege • 6-6
 BYPASS • 7-6
 defined by access mode • 2-3
 I/O operations • 7-2
 logical I/O • 7-4, 7-7
 MOUNT • 7-4
 physical I/O • 7-4, 7-7
 setting for process • SYS-372
 SYSTEM • 7-6
 user • 2-2

Privileged shareable image
 See User-written system service

Process

 creating • 8-2, SYS-77
 creation restriction • 8-7
 deleting • 8-16, SYS-122
 detached • 8-2, 8-7
 disabling swap mode • 11-7
 disallowing swapping • 11-7
 getting information about
 asynchronously • SYS-209
 synchronously • SYS-222.2
 hibernating • 8-11, SYS-296
 identification • 8-7
 information • 8-9
 name • 8-7
 name within group • 8-8
 quota
 symbolic names for (PQL\$_xxxx) • SYS-80
 resuming after suspension • SYS-347
 scheduling wakeup for • SYS-353
 setting name of • SYS-368
 setting priority of • SYS-366
 setting privilege • SYS-372
 setting swap mode for • SYS-384
 subprocess • 8-2
 suspending • 8-11, 8-13, SYS-443
 swapping • 11-7
 swapping by suspension • 8-13
 termination mailbox • 7-33, 8-19
 waiting for entire set of event flags • SYS-466
 waiting for event flag to be set • SYS-463
 waiting for one of set of event flags • SYS-468
 waking • SYS-464
Process directory table • 6-3
Process identification number
 See PID
Process index number • SYS-217
Process logical name table • 6-4
Process rights list • 3-2

Program region • 11-2
 adding page to • SYS-153
 deleting page from • SYS-59
 Protected shareable image
 See User-written system service
 Protection
 by access mode • 2-3
 device • 7-6
 I/O operations • 7-2
 mailbox • 7-4, 7-5
 page • 11-5
 queue • SYS-428.3
 setting for page • SYS-369
 volume • 7-4
 Protection mask • 7-4

Q

Queue
 creating and managing
 asynchronously • SYS-393
 synchronously • SYS-428.8
 getting information about
 asynchronously • SYS-239
 lock management • 12-4
 protection • SYS-428.3
 types of • SYS-428.1, SYS-444
 Quota
 AST • 7-3
 buffered I/O • 7-3
 buffered I/O byte count • 7-3
 direct I/O • 7-3
 establishing • 6-8
 I/O operations • 7-2
 process
 symbolic names for (PQL\$_xxxx) • SYS-80
 resource • 2-2

R

Remote node
 establishing logical link with • SYS-20
 Resource
 controlling • 8-6
 lock management concept • 12-1
 name • 12-2
 quota • 2-2
 RESOURCE attribute • 3-4

Resource wait mode • 2-2
 setting • SYS-376
 Return address array • 11-4
 Return condition
 special • 2-13
 Return condition value • 2-14
 high-level language • 2-17
 Rights database • 3-2, 3-5, 3-14
 adding to • 3-8
 default protection • 3-6
 elements of • 3-6
 holder record • 3-5
 identifier record • 3-5
 initializing • 3-6
 keys • 3-5
 modifying • 3-12, 3-14
 Rights list • 3-26
 RMS (Record Management Services)
 See VAX RMS

S

Sample program • 13-1
 Searching operations • 3-14
 Search list • 6-2
 Section • 11-8
 characteristic • 11-10
 creating • 11-8, SYS-96
 defining extent • 11-10
 deleting • 11-18
 deleting global • SYS-130
 global paging file • 11-16
 image • 11-19
 mapping • 11-14, SYS-96
 page frame • 11-19
 paging • 11-16, 11-17
 unmapping • 11-18
 using to share data • 11-18
 writing back • 11-19
 writing modifications to disk • SYS-458, SYS-462
 Section file
 updating • SYS-458, SYS-462
 Service routine
 AST • 5-4
 Signal array argument • 10-9
 Stack limit
 changing size of • SYS-382
 Stack pointer
 adjusting • SYS-5

Index

String

- formatting output • SYS-155
- searching for file specification in • SYS-169
- Subblock • 12-10
- Subprocess • 8-2, SYS-89
 - disk and directory default • 8-5
 - image • 8-3
 - input, output, and error device • 8-3
- Suspension • 8-11, 8-13
 - compared with hibernation • 8-11
- Symbolic definition macro • 2-8
- SYS\$ADD_HOLDER • 3-8, SYS-1
- SYS\$ADD_IDENT • 3-8, SYS-3
- SYS\$ADJSTK • SYS-5
- SYS\$ADJWSL • 11-6, SYS-7
- SYS\$ALLOC • SYS-9
 - example • 7-21
- SYS\$ASCEFC • SYS-12
- SYS\$ASCTIM • SYS-15
 - example • 9-2
- SYS\$ASCTOID • 3-7, SYS-18
- SYS\$ASSIGN • SYS-20
 - example • 7-13
- SYS\$BINTIM • 9-3, SYS-24
- SYS\$BRKTHRU • SYS-27
- SYS\$BRKTHRUW • SYS-35
- SYS\$CANCEL • SYS-36
 - example • 7-20
- SYS\$CANEXH • SYS-38
- SYS\$CANTIM • SYS-39
 - example • 9-6
- SYS\$CANWAK • 9-6, SYS-41
- SYS\$CHANGE_ACL • 3-16, 3-23, SYS-43
- SYS\$CHECK_ACCESS • 3-28, SYS-46.1
- SYS\$CHFDEF macro • 10-9
- SYS\$CHKPRO • 3-27, SYS-47
- SYS\$CLREF • 4-4, SYS-54
- SYS\$CMEXEC • SYS-55
- SYS\$CMKRNL • SYS-57
- SYS\$CNTREG • SYS-59
 - See also SYS\$DELTVA
- SYS\$CREATE_RDB • 3-6, SYS-91
- SYS\$CRELNM • SYS-61
- SYS\$CRELNT • SYS-66
- SYS\$CREMBX • SYS-72
- SYS\$CREPRC • SYS-77
 - example • 8-3
- SYS\$CRETVA • SYS-93
 - See also SYS\$EXPREG
- SYS\$CRMPSC • SYS-96
- SYS\$DACEFC • SYS-106
- SYS\$DALLOC • SYS-107
- SYS\$DASSGN • SYS-109
 - example • 7-18
- SYS\$DCLAST • SYS-111
 - example • 5-5
- SYS\$DCLCMH • SYS-113
- SYS\$DCLEXH • SYS-115
 - example • 8-15
- SYS\$DELLNM • SYS-117
- SYS\$DELMBX • SYS-120
- SYS\$DELPRC • 8-17, SYS-122
- SYS\$DELTVA • SYS-124
- SYS\$DEQ • SYS-126
 - example • 12-13
- SYS\$DGBLSC • SYS-130
- SYS\$DISMOU • 7-24, SYS-133
- SYS\$DLCEFC • SYS-136
- SYS\$ENQ • SYS-138
 - example • 12-7, 12-10
- SYS\$ENQW • SYS-148
- SYS\$ERAPAT • 3-28.1, SYS-150
- SYS\$EXIT • 8-14, SYS-152
 - causing call to for process • SYS-181
- SYS\$EXPREG • SYS-153
 - example • 11-2
- SYS\$FAO • SYS-155
 - directive
 - format of • SYS-157
 - list of • SYS-158
 - example • 7-27, SYS-162, SYS-167
- SYS\$FAOL
 - example • SYS-164
- SYS\$FILESCAN • SYS-169
- SYS\$FIND_HELD • 3-9, 3-14, SYS-174
- SYS\$FIND_HOLDER • 3-9, 3-14, SYS-177
- SYS\$FINISH_RDB • SYS-180
- SYS\$FORCEX • SYS-181
 - See also SYS\$DELPRC
 - example • 8-15
- SYS\$FORMAT_ACL • 3-16, 3-22, SYS-183
- SYS\$GETDVI • SYS-192
- SYS\$GETDVIW • SYS-208
- SYS\$GETJPI • SYS-209
 - example • SYS-222
- SYS\$GETJPIW • SYS-222.2
- SYS\$GETLKI • SYS-223
- SYS\$GETLKIW • SYS-234
- SYS\$GETMSG • SYS-235
- SYS\$GETQUI • SYS-239
- SYS\$GETSYI • SYS-272
- SYS\$GETSYIW • SYS-282
- SYS\$GETTIM • 9-2, SYS-283
- SYS\$GETUAI • SYS-284

- SYS\$GRANTID • SYS-292
 SYS\$HIBER • SYS-296
 example • 8-12
 SYS\$IDTOASC • 3-7, 3-14, SYS-298
 SYS\$LCKPAG • SYS-301
 SYS\$LKWSET • 11-6, SYS-303
 SYS\$MGBLSC • SYS-305
 SYS\$MOD_HOLDER • 3-12, SYS-309
 SYS\$MOD_IDENT • 3-12, SYS-312
 SYS\$MOUNT • 7-22, SYS-315
 SYS\$MTACCESS • 3-28.1, SYS-320.6
 SYS\$NUMTIM • 9-7, SYS-321
 SYS\$PARSE_ACL • 3-16, 3-22, SYS-323
 SYS\$PURGWS • SYS-325
 See also SYS\$ADJWSL
 SYS\$PUTMSG • SYS-326
 SYS\$QIO • SYS-334
 example • 7-13
 SYS\$QIOW • SYS-340
 SYS\$READEF • SYS-341
 SYS\$REM_HOLDER • 3-14, SYS-343
 SYS\$REM_IDENT • 3-14, SYS-345
 SYS\$RESUME • SYS-347
 SYS\$REVOKID • SYS-349
 SYS\$RMSRUNDN • SYS-446.1
 SYS\$SCHDWK • SYS-353
 canceling • 9-6
 converting time format for • SYS-24
 example • 9-6
 request • 9-6
 SYS\$SETAST • SYS-356
 SYS\$SETDDIR • SYS-446.3
 SYS\$SETDFPROT • SYS-446.5
 SYS\$SETEF • 4-4, SYS-357
 SYS\$SETEXV • SYS-358
 example • 10-7
 SYS\$SETIME • 9-7, SYS-360
 SYS\$SETIMR • 9-4, SYS-362
 converting time format for • SYS-24
 example with AST • 5-1
 SYS\$SETPRA • SYS-364
 SYS\$SETPRI • SYS-366
 SYS\$SETPRN • SYS-368
 SYS\$SETPRT • SYS-369
 SYS\$SETPRV • SYS-372
 SYS\$SETRWM • 7-2, SYS-376
 SYS\$SETSFM • SYS-378
 example • 2-13
 SYS\$SETSSF • SYS-380
 SYS\$SETSTK • SYS-382
 SYS\$SETSWM • SYS-384
 example • 11-7
 SYS\$SETUAI • SYS-385
 SYS\$SNDERR • SYS-393
 SYS\$SNDJBC • SYS-393
 SYS\$SNDJBCW • SYS-428.8
 SYS\$SNDOPR • SYS-429
 SYS\$SUSPND • SYS-443
 SYS\$SYNCH • SYS-445
 SYS\$TRNLNM • SYS-447
 SYS\$ULKPAG • SYS-452
 SYS\$ULWSET • SYS-454
 SYS\$UNWIND • SYS-456
 example • 10-15
 SYS\$UPDSEC • SYS-458
 SYS\$UPDSECW • SYS-462
 SYS\$WAITFR • SYS-463
 SYS\$WAKE • SYS-464
 See also SYS\$HIBER
 example • 8-12
 SYS\$WFLAND • SYS-466
 SYS\$WFLOR • SYS-468
 SYS\$PRV • 7-6
 System
 exception dispatcher • 10-7
 getting information about
 asynchronously • SYS-272
 synchronously • SYS-282
 library • 2-1, 2-5
 mailbox • 7-32
 message • 2-16
 System directory table • 6-3
 System logical name table • 6-6
 System service
 checking completion status of • SYS-445
 failure exception condition • 2-13
 inhibiting user mode calls to • SYS-380
 macro • 2-1, 2-5
 setting failure exception mode • SYS-378
 setting filter • SYS-380
 System time
 setting • SYS-360
-
- ## T
-
- Tape volume
 mounting • 7-22
 Terminal I/O
 example • 7-18
 Termination mailbox • 7-33, 8-19
 Termination message
 format • SYS-86

Index

Time

- absolute • 9-2
- conversion • 9-1
- converting ASCII to binary • 9-3
- converting binary to ASCII string • SYS-15
- converting binary to numeric • SYS-321
- delta • 9-2
- getting current system • 9-2, SYS-283
- numeric and ASCII • 9-7
- setting system • 9-7, SYS-360
- system format • 9-2

Timer

- setting • SYS-362

Timer request • 9-4

- canceling • 9-6, SYS-39

TQELM (timer queue entry limit) quota

- effect of canceling timer request • SYS-40

U

UAF (user authorization file)

- getting information about • SYS-284
- modifying • SYS-385

User-defined logical name tables • 6-6

User privilege • 2-2

User-written system service • A-1

V

VAX RMS (Record Management Services) • 7-1

- opening file for mapping • 11-9

Virtual address space • 11-2, 11-3

- adding page to • SYS-93, SYS-153
- creating • SYS-93
- deleting page from • SYS-59, SYS-124
- increasing and decreasing • 11-2
- layout • 11-2
- mapping section of • 11-14
- specifying array • 11-5

Virtual I/O • 7-7

- canceling requests for • SYS-36

VMS usage • 1-6

Volume

- dismounting • SYS-133
- getting information about
 - asynchronously • SYS-192
 - synchronously • SYS-208
- mounting • 7-22, SYS-315

Volume protection • 7-4

W

Wakeup

- canceling • SYS-41
- scheduling • 9-6

Working set

- adjusting limit • SYS-7
- adjusting size • 11-6
- locking page into • 11-6, SYS-303
- paging • 11-6
- purging • SYS-325
- unlocking page from • SYS-454

Write back section • 11-19

READER'S COMMENTS

Note: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well organized? Please make suggestions for improvement.

Did you find errors in this manual? If so, specify the error and the page number.

Please indicate the type of user/reader that you most nearly represent:

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify) _____

Name _____ Date _____

Organization _____

Street _____

City _____ State _____ Zip Code _____
or Country

Do Not Tear - Fold Here and Tape

digital



No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

SSG PUBLICATIONS ZK1-3/J35
DIGITAL EQUIPMENT CORPORATION
110 SPIT BROOK ROAD
NASHUA, NEW HAMPSHIRE 03062-2698



Do Not Tear - Fold Here

Cut Along Dotted Line

NOTES

NOTES

NOTES

NOTES

NOTES

NOTES

